Horváth Gábor

ASSEMBLY VÉDETT MÓDÚ PROGRAMOZÁS

LSI Oktatóközpont

SZÁMALK

Horváth Gábor

ASSEMBLY
VÉDETT MÓDÚ PROGRAMOZÁS

lemezmelléklettel

Nyitott rendszerű képzés – Távoktatás –
Oktatási segédlete
Felsőoktatási Tankönyv

LSI Oktatóközpont
A Mikroelektronika Alkalmazásának
Kultúrájáért Alapítvány

Budapest, 1998

Készült a Művelődési és Közoktatási Minisztérium támogatásával

Lektorálta: Heltmann Attila
Ribényi András

ISBN 963 577 199 1

Kiadó: **LSI Oktatóközpont**
Felelős vezető: **Dr. Kovács Magda**
Témafelelős: **Flier István**

OLITON Kft.
Bp. Vajda Péter u. 12.

TARTALOMJEGYZÉK

ELŐSZÓ 9

I. RÉSZ 11

1. A VÉDETT MÓDÚ PROGRAMOZÁS ELMÉLETI ALAPJAI 11
1.1. Bevezetés 11
1.2. Az új és megváltozott regiszterek 13
1.3. A védett módú memóriacímzés 18
1.4. A deszkriptorok 19
1.5. A deszkriptortáblázatok 22
1.6. A szelektorok 23
1.7. Memóriacímzés egy példán keresztül 23
1.8. A 16 és 32 bites szegmensek 25

2. VÉDELEM A VÉDETT MÓDBAN 28
2.1. Típusellenőrzés 28
2.2. Határellenőrzés 29
2.3. Privilégiumszintek 30
2.3.1. A privilégiumszintek ellenőrzése 31
2.3.2. A privilégiumszintek – Az adatok 31
2.3.3. A privilégiumszintek – A verem 32
2.3.4. A privilégiumszintek – A kódszegmensek 32
2.3.5. Vezérlésátadás másik kódszegmensre 33
2.3.6. Vezérlésátadás call-kapukon keresztül 33
2.3.7. A RETF utasítás 35
2.4. Privilegizált és érzékeny utasítások 35

3. A VÉDETT MÓDÚ PROGRAMOZÁS MAGAS SZINTEN 38
3.1. Megszakítások és kivételek 38
3.1.1. A megszakítások működésének bemutatása egy konkrét példán keresztül 40
3.1.2. Kivételek 41
3.2. Multitaszking 46
3.2.1. A multitaszking operációs rendszerek típusai 46
3.2.2. A hardver és a szoftver multitaszking 48
3.2.3. A taszk-állapotszegmens 48
3.2.4. A TSS deszkriptor 51

3.2.5. A taszkregiszter ... 52
3.2.6. Taszkkapuk ... 52
3.2.7. A taszkkapcsolás ... 53
3.3. Lapozás ... 55
3.3.1. Lapfordítás ... 56
3.3.2. Laptáblázat bejegyzések ... 58
3.3.3. A laphiba ... 59
3.3.4. A lapozás jellemzői ... 60
3.4. A virtuális 8086-os üzemmód ... 62
3.4.1. A virtuális üzemmódú taszkok elindítása ... 63
3.4.2. A virtuális módú környezet emulálása ... 63
3.4.3. Az IF flag emulálása ... 64
3.4.4. A megszakítások működése és emulálása ... 65

4. A PROCESSZOR FELISMERÉSE ... 69
4.1. A 386-os processzor és a virtuális mód felismerése ... 69
4.2. A 486-os és Pentium processzorok felismerése ... 70
4.3. A CPUID utasítás használata ... 72

II. RÉSZ ... 76

1. EGYSZERŰ VÉDETT MÓDÚ PROGRAMOK ... 76
1.1. Átkapcsolás védett módba ... 76
1.1.1. A globális deszkriptortáblázat beállítása ... 76
1.1.2. A szegmensek közötti ugrások ... 79
1.1.3. Átkapcsolás védett módba és vissza ... 80
1.2. Adatszegmens elérése ... 83
1.3. A veremszegmens ... 86
1.4. A képernyőszegmens elérése ... 89
1.5. A kivételek kezelése ... 91
1.5.1. Az IDT beállítása ... 91
1.5.2. A kivételkezelők ... 92
1.6. A megszakítások kezelése ... 96
1.6.1. A programozható megszakítás-vezérlő ... 96
1.6.2. Az IRQ-k áthelyezése ... 97
1.6.3. IRQ 0 – időzítés ... 100
1.7. Képernyő kezelése védett módban I. ... 101
1.8. Képernyő kezelése védett módban II. ... 104
1.9. A regiszterek kiírása a képernyőre kivétel esetén ... 107

2. A TASZKOK ÉS A VÉDELEM ... 116
2.1. A szegmensszintű védelem ... 116
2.1.1. Szimbólum-definíciók ... 118
2.1.2. Az új GDT ... 120
2.1.3. Az új IDT ... 122

2.2. Taszkváltás egy 0. szintű taszkra 132
2.3. Taszkváltás egy 3. szintű taszkra 138
2.4. Funkciók hívása megszakításokon keresztül 143
2.5. Két darab 3. privilégiumszintű taszk váltogatása 146
2.5.1. A globális deszkriptorok veszélyei 150
2.6. Lokális deszkriptorok 153
2.7. Privilégiumszintek és I/O utasítások 160
2.8. I/O műveletek bittérképpel 165
2.9. Illeszkedő kódszegmensek és call-kapuk 170

3. LAPOZÁS ÉS A VIRTUÁLIS ÜZEMMÓD 175
3.1. A lapozás engedélyezése, a táblázatok beállítása 175
3.2. Egy taszk elhelyezkedése az 1, 2 és 3 GB-os területen 180
3.3. Több taszk azonos címterületen 189
3.4. Egy virtuális üzemmódú taszk elindítása 197
3.5. Valós módban 4 GB memória elérése 200

III. RÉSZ 204

1. EXTENDED MEMORY SPECIFICATION (XMS) 204
1.1. Az XMS driver ellenőrzése 204
1.2. Az XMS funkciók 204
1.3. Az XMS driver információ lekérdezése 206
1.4. A HMA-val kapcsolatos funkciók 206
1.5. Az A20 funkciók 206
1.6. Kiterjesztett memória kezelése 207
1.7. Az UMB-vel kapcsolatos funkciók 210
1.8. 32 bites kiterjesztett memória funkciók 210
1.9. Az XMS hibakódok 211
1.10. Példa a kiterjesztett memória használatára védett módban 212

2. DOS PROTECTED MODE INTERFACE (DPMI) 217
2.1. Átkapcsolás védett módba a DPMI segítségével 217
2.2. A DPMI funkciók hívása 220
2.3. Lokális deszkriptorok kezelése 220
2.4. Memóriakezelő funkciók 226
2.5. Lapozással kapcsolatos funkciók 228
2.6. A megszakításokkal kapcsolatos funkciók 230
2.7. Valós és védett mód közötti váltások 232
2.8. DPMI hibakódok 238

ELŐSZÓ

E könyvnek az a célja, hogy bemutassa a 32 bites védett módú programozást, elsősorban gyakorlati szemszögből.

Mivel a védett módú programok a felhasználásukat és az összetettségüket tekintve nagyon sokfélék lehetnek, ezért a könyv példáinak kialakításakor sem egy komplett, univerzális DOS extender megalkotása volt a cél (ezekből már számtalan létezik), hanem a védett módú programozás alapjainak bemutatása – egyszerű példaprogramokon keresztül.

A könyv első része a védett mód elméleti alapjait röviden összefoglalva tárgyalja. (A 386/486-os, és a Pentium processzorok részletes ismertetése megtalálható más könyvekben.) A második részben a védett mód számos lehetőségéről lesz szó – gyakorlati példákon keresztül. Az egyszerű példaprogramok több témakört ölelnek fel, többek között a védett módba kapcsolást, a védelmet, a taszkváltásokat, a lapozást és a virtuális üzemmódot. Végül pedig az XMS és a DPMI ismertetésére kerül sor, amelyek megkönnyítik a védett módú programok írását.

A könyv és a benne szereplő példák megértésének feltétele az assembly nyelv ismerete, illetve a programozásban való jártasság.

Horváth Gábor

I. RÉSZ

1. A védett módú programozás elméleti alapjai

1.1. Bevezetés

A védett módú programozásban egy új korszak kezdődött a 80386-os processzor megjelenésével, amely már egy valódi 32 bites processzor. Bár a védett mód a 80286-os processzornál jelent meg először, viszont ez 16 bites, és a megcímezhető memória maximális mérete is csak 16 MB volt (a 8086-nál pedig 1 MB). A 80386-os esetén 4 GB memória érhető el, továbbá a processzor alkalmassá vált például a virtuális memória és a multitaszking biztonságos kezelésére is. (A későbbiekben csak a 386-os és jobb processzorokról lesz szó.)

A 80386-os processzor az alábbi 3 üzemmódban képes működni, és ezek között szabadon tud váltogatni:

- Valós mód (real mode vagy real-address mode)
- Védett mód (protected mode)
- Virtuális 8086-os mód

A korábbi processzorokkal való kompatibilitás miatt a gép bekapcsolása (vagy RESET) után a CPU valós módban kezdi meg a végrehajtást, amelynek működése hasonlít a 8086-osra, vagyis nincsen védelem, és a megcímezhető memória 1 MB. A védett mód a processzor természetes üzemmódja, az új szolgáltatások (védelem, lapozás, multitaszking, stb.) ebben a módban érhetők el. A virtuális 8086-os mód használatával pedig a régi processzorokra írt programok futtathatóak, anélkül, hogy valós módba kellene kapcsolni. Ekkor a programok a védett módú rendszer felügyelete alatt futnak, és így lehetővé válik egyszerre több 8086-os alkalmazás biztonságos futtatása is.

Az egyre több memória elérésének igénye miatt sokféle protokoll keletkezett. Ezeknek a célja az volt, hogy megkönnyítsék a memóriakezelést, de sajnos a kompatibilitás és egyéb tényezők miatt inkább megnehezítették.

Kezdetben megjelent az expanded memóriakártya, amely lehetővé tette a 640 KB-nál több memória használatát. Amikor az extended (kiterjesztett) memória kezdett elterjedni, már sok program létezett, amely az expanded memóriát használta. Ezért a kompatibilitás miatt megszületett néhány memória-kezelő program (pl. QEMM, EMM386), amelyek szoftveresen emulálták az expanded memóriát.

A 80386-os processzorral számtalan DOS Extender született, amelyek a valós módú DOS szolgáltatásait elérhetővé tették védett módban is, viszont az egyes programok nem voltak kompatibilisek egymással. Ezután kezdtek megjelenni az új protokollok. Először a VCPI (Virtual Control Program Interface), amely továbbra is az expanded memóriát emulálta, de lehetővé tette a védett módba kapcsolást a 0. privilégiumszintre. Az XMS (eXtended Memory Specification) végre egységesítette a 640 KB feletti területek használatát, de elsősorban csak a valós módú programok számára tette elérhetővé a kiterjesztett memóriát (és csak adatok tárolására).

A védett módú funkcióhívások (memóriakezelés, megszakítások, stb.) szabványosítását a DPMI (DOS Protected Mode Interface) bevezetése jelentette, amelyet a Microsoft hozott létre a Windows melléktermékeként. Bár a DPMI alatt futó programok valamivel lassabban futnak (elsősorban a 3. privilégiumszint és az emulálások miatt), viszont a DPMI szolgáltatásait használó programok hordozhatóvá váltak, mivel már sok extender és operációs rendszer támogatja ezeket a funkciókat.

A védett mód alapjainak áttekintése után szereplő példákban elsősorban saját védett módú környezet kialakítása a cél, de található több példa a DPMI-vel és az XMS-el kapcsolatban is.

1.2. Az új és megváltozott regiszterek

A 8086-os processzorhoz képest a 80386-os általános regiszterei, a veremmutatói és az utasításmutatója 32 bitesre bővült ki. A 32 bites regiszterek az EAX, EBX, ECX, EDX, ESI, EDI, illetve ESP, EBP és EIP. Ezeknek az alsó 16 bites részük (AX, BX, CX, DX, SI, DI, SP, BP és IP), valamint az első 4 regiszter alsó két byte-ja (AL, AH, BL, BH, CL, CH, DL, DH) továbbra is használható önálló regiszterként. A 32 bites regiszterek felső része nem érhető el közvetlenül, hanem csak eltolásokkal vagy egyéb műveleteken keresztül.

A szegmensregiszterek mérete nem növekedett, vagyis változatlanul 16 bitesek. A funkcióik sem változtak meg, tehát a CS a kódszegmensre, az SS a veremszegmensre, a DS az adatszegmensre, az ES, illetve a két új szegmensregiszter, az FS és a GS pedig egyéb szegmensekre mutat. Védett üzemmódban a szegmensregiszterek felépítése viszont eltér a valós módhoz képest, mivel már nem az adott szegmens báziscímét tartalmazzák, hanem egy táblázaton belüli elemre mutatnak, ahol a szegmens 32 bites báziscíme és egyéb információk találhatóak.

Az eddigi 16 bites flagregiszter is 32 bitesre bővült (EFLAGS), és kiegészült néhány rendszerflaggel.

A EFLAGS 0–11. közötti bitjei változatlanok maradtak, a 12. bittől kezdve a flagek csak védett módban használatosak. Az AC bit 80486-os processzoron jelent meg először, a 18. feletti bitek pedig csak a Pentium és jobb processzorokon érvényesek.

Az EFLAGS 12. és 13. bitje határozza meg, hogy az adott taszk mely privilégiumszinteken használhat I/O utasításokat. Ezzel például korlátozható az alkalmazói programok számára a CLI, STI, IN, OUT, stb. utasítások használata.

Ha egy taszk FAR CALL utasítással vagy egy megszakításon keresztül hívódott meg, akkor az NT bit 1-re állítódik, különben mindig 0. Ez a bit az IRETD utasítás működését befolyásolja. Ha egy beágyazott taszkban (NT = 1) kerül végrehajtásra az IRETD, akkor egy taszkkapcsolás történik a

hívó taszkra, egyébként pedig (ha NT = 0) a veremben lévő CS:EIP-re adódik vissza a vezérlés.

0	CF	átvitel flag (Carry Flag)
1	1	fenntartva, mindig 1
2	PF	paritás flag (Parity Flag)
3	0	fenntartva
4	AF	fél-átvitel flag (Auxiliary Carry Flag)
5	0	fenntartva
6	ZF	zéró flag (Zero Flag)
7	SF	előjel flag (Sign Flag)
8	TF	csapda flag (Trap Flag)
9	IF	megszakítás flag (Interrupt Flag)
10	DF	irányjelző flag (Direction Flag)
11	OF	túlcsordulás flag (Overflow Flag)
12–13	IOPL	I/O privilégiumszint (I/O Privilege Level)
14	NT	beágyazott taszk (Nested Task)
15	0	fenntartva
16	RF	folytatás flag (Resume Flag)
17	VM	8086 virtuális üzemmód (Virtual Mode)
18	AC	illeszkedés ellenőrzés (Alignment Check)
19	VIF	virtuális megszakítás flag (Virtual Interrupt Flag)
20	VIP	függőben lévő virtuális megszakítás (V.I. Pending Flag)
21	ID	CPU azonosító flag (Identification Flag)
22–31	–	fenntartva

1.1. ábra Az EFLAGS regiszter bitjei

Az RF bit segítségével elérhető, hogy egyetlen utasítás végrehajtása közben ne keletkezzen többször nyomkövetési hiba.

Ha egy taszk VM bitje magasra van állítva, akkor az virtuális üzemmódban fut a védett módú felügyelőprogram irányítása alatt. A VM bit csak taszkkapcsolással vagy az IRETD utasítással befolyásolható.

Az AC flaggel a memóriaoperandusok megfelelő illeszkedése ellenőrizhető. Az illeszkedési kivétel csak a 3. privilégiumszinten keletkezik, ha az AC és a CR0 regiszterben lévő AM (Alignment Mask) bitek értéke 1.

A VIP és a VIF flagek segítségével az alkalmazói programok számára a megszakítások engedélyezése és letiltása virtuálissá tehető. Ekkor a CLI és az STI utasítások nem az IF bitet változtatják meg közvetlenül.

Az ID flag az újabb processzorokon megkönnyíti a CPU-k felismerését és pontos azonosítását. Erről bővebben egy későbbi fejezetben lesz szó.

A processzor működését az EFLAGS-on kívül egyéb regiszterek is befolyásolják. Ezek a vezérlőregiszterek (Controll Registers) a CR0, CR1, CR2, CR3 és CR4. (Az opkódok alapján 8 darab vezérlőregiszter lehetséges.) A CR0 regiszterben helyezkednek el a legfontosabb bitek, amelyek lehetővé teszik a védett és a valós mód közötti váltást, a lapozást, illetve a koprocesszor emulálását, stb. A CR2 és a CR3 regisztereknek a lapozásban van szerepük. Laphiba esetén a CR2-ben található a kivételt okozó memóriacím lineáris címe. A CR3 az aktuális taszk lapcímtárának helyét (lényegében a felső 20 bitjét) tartalmazza. A CR4 regiszter pedig a Pentium és a későbbi processzorok kiterjesztéseinek engedélyezésére szolgál.

A vezérlőregisztereket a MOV CRn,reg illetve a MOV reg,CRn utasításokkal lehet elérni. Védett módban ezek privilegizált utasítások, tehát csak a 0. privilégiumszinten hajthatóak végre, valós módban viszont nincs ilyen korlátozás. A CR0 regiszter alsó 16 bitje elérhető az SMSW és az LMSW utasításokkal is (Store & Load Machine Status Word).

A következő táblázat a CR0 egyes bitjeit mutatja be (az 5–30. bitek csak 80486-tól kezdve érvényesek), ezek közül szinte csak a legalsó és a legfelső biteket használjuk a könyvben szereplő példákban. A védett módba kapcsoláshoz a PE bitet 1-re kell állítani, a valós módba való visszatéréshez pedig törölni kell. A 31. bittel a lapozást lehet engedélyezni, illetve letiltani.

0	PE	Védett mód engedélyezése; ha 1, akkor a processzor védett módban van, ha 0, akkor valós módban. (Protection Enable)
1	MP	Ha 1, akkor a koprocesszor jelen van (a TS bit tesztelése FPU utasításoknál), ha 0, akkor nincs jelen. (Math Present vagy Monitor coProcessor)
2	EM	Numerikus koprocesszor emulálása; ha 1, akkor a koprocesszor utasítások 7. kivételt okoznak. (Emulation)
3	TS	Taszkkapcsolás bit; taszkváltások során 1-re állítódik, és ekkor minden FPU utasítás 7. kivételt okoz. (Task Switched)
4	ET	Koprocesszor bővítés; jelzi a koprocesszor típusát, 1 = 32 bites 80387, 0 = 16 bites 80287. (Extension Type)
5	NE	Numerikus hiba bit; a lebegőpontos kivételek létrejöttét befolyásolja. (Numeric Error)
6–15	–	Fenntartva
16	WP	Írásvédelem bit; ha 0, akkor a felügyelői szinten az írásvédett lapokba is lehet írni, ha 1, akkor pedig nem. (Write Protect)
17	–	Fenntartva
18	AM	Illeszkedés maszk; ha 1, akkor az illeszkedés ellenőrzés engedélyezve van (az AC bittől is függ). (Aligment Mask)
19–28	–	Fenntartva
29	NW	Átírástiltás bit; ha 0, akkor a cache átírás engedélyezve van, ha 1, akkor le van tiltva. (Not Write-through)
30	CD	Cache letiltás bit, ha 1, akkor a cache használaton kívül van, ha 0, akkor a cache-be írás engedélyezett. (Cache Disable)
31	PG	A lapozás engedélyezése; ha 1, akkor a lapozás engedélyezve van, ha 0, akkor le van tiltva. (Paging Enable)

1.2. ábra A CR0 vezérlőregiszter

A következő táblázatban a CR4 regiszter bitjei szerepelnek. Ezeket csak a Pentium és az újabb processzorok támogatják, és bizonyos kiterjesztéseket engedélyeznek vagy tiltanak le. Ezek a kiterjesztések elég hasznosak, például a VME bit magasra állításával a virtuális módú taszkok akár 15–20-szor gyorsabban is futhatnak. A kiterjesztések jelenlétének vizsgálatához a később tárgyalandó CPUID utasítás használható fel.

0	VME	Ha 1, akkor virtuális üzemmódban a virtuális megszakítások engedélyezve vannak, ha 0, akkor nem. (Virtual Mode Extensions)
1	PVI	Ha 1, akkor védett üzemmódban a virtuális megszakítások engedélyezve vannak, ha 0, akkor nem. (Protected Mode Virtual Interrupts)
2	TSD	Ha 0, akkor RDTSC utasítás minden szintről elérhető, ha 1, akkor csak a 0. privilégiumszintről. (Time Stamp Dissable)
3	DE	Az I/O töréspontok engedélyezése (DE = 1), vagy letiltása (DE = 0). (Debug Extension)
4	PSE	Ha 1, akkor a 4 MB-os lapméret engedélyezve van, ha 0, akkor a lapok mérete a szokásos 4 KB. (Page Size Extension)
5	PAE	Ha 1, akkor a fizikai cím kiterjesztés engedélyezve van (PentiumPro), ha 0, akkor le van tiltva. (Page Address Extension)
6	MCE	Ha 1, akkor a processzortípus ellenőrzés kivétel (18. kivétel) engedélyezett, ha 0, akkor nem. (Machine Check Extension)
7	PGE	Ha 1, akkor a globális lapok engedélyezettek (PentiumPro), ha 0, akkor nem. (Page Global Enable)
8	PCE	Ha 0, akkor RDPMC utasítás minden szintről elérhető, ha 1, akkor csak a 0. privilégiumszintről. (Performance Counter Enable)
9–31	–	Fenntartott

1.3. ábra A CR4 vezérlőregiszter

A nyomkövetés megkönnyítésére szolgálnak a DR0–DR7 regiszterek, amelyek segítségével például kód-, és adattöréspontok egyaránt elhelyezhetőek, és a kódszegmenset sem kell módosítani (INT 3, stb.).

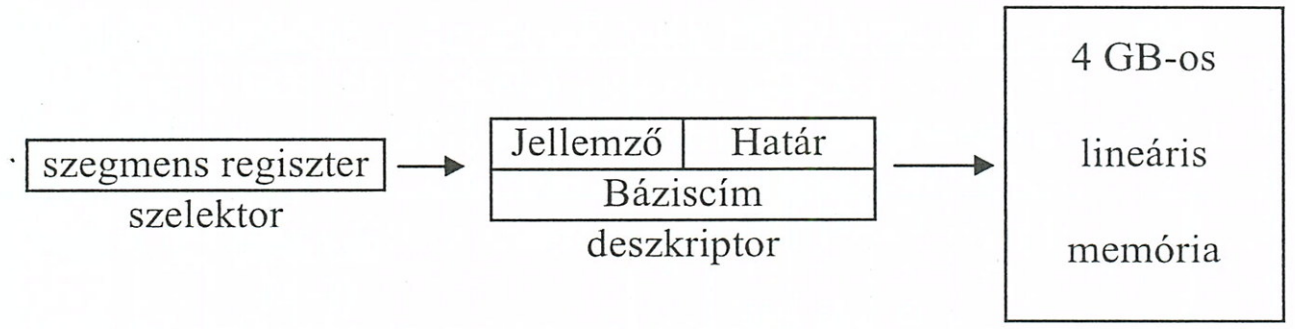
Az újabb processzorok már 64 bites modell-specifikus regisztereket (Model Specific Registers) is tartalmaznak. Ezek az RDMSR és az WRMSR (valamint az RDTSC és az RDPMC) utasításokkal érhetőek el. Az MSR-ek közé tartoznak például a processzor ellenőrzésével kapcsolatos regiszterek (MCA, MCT), az új tesztregiszterek (a régi TR0-TR7 helyett), és a Time Stamp Counter (TSC). Ezek használata számos új lehetőséget nyújt, például a TSC regiszter az utolsó reset óta eltelt óraciklusokat számolja, és akár több ezer éven keresztül pontos értéket tartalmaz. Ezáltal bizonyos utasítások és eljárások végrehajtásához szükséges órajelek száma nagyon pontosan megállapítható. A megvalósított regiszterek pontos elhelyezkedése és funkciói viszont eltérőek lehetnek az egyes processzortípusoknál és a különböző gyártók esetén.

1.3. A védett módú memóriacímzés

Minden memóriahivatkozás egy báziscímből és egy ofszetcímből áll, függetlenül attól, hogy utasítás, adat, vagy verem címére vonatkozik-e, illetve, hogy valós vagy védett módban vagyunk-e.

Valós módban a szegmensregiszterekben lévő érték a 20 bites lineáris cím felső 16 bitjét tartalmazza, és ehhez adódik hozzá a 16 bites ofszetcím. Így a 20 biten megcímezhető maximális memória mérete nagyjából 1 MB, és egy szegmensregiszterrel egyszerre csak 64 KB-ot lehet megcímezni.

Mivel védett módban a szegmensregiszterek továbbra is 16 bitesek, ezért más módszer szükséges ahhoz, hogy több memóriát lehessen megcímezni. Védett módban a szegmensregiszterek nem közvetlenül a báziscímet tartalmazzák, hanem ún. szelektorokat, amelyek egy táblázatban lévő deszkriptort (leírót) választanak ki. Ez a deszkriptor tartalmazza az aktuális szegmens báziscímét, méretét, és egyéb jellemzőit. A 8 byte-os deszkriptoron belül a báziscím már 32 bites, ezáltal lényegesen több memória címezhető meg.



1.4. ábra A védett módú címzésmód vázlatos rajza

Valós módban a szegmensregiszterekkel egyszerre 64 KB méretű területet lehetett megcímezni (se többet, se kevesebbet). Védett módban viszont már a szegmensek mérete is beállítható, erre szolgál a deszkriptorban lévő 20 bites határ. A méret alapállapotban tehát újra csak 1 MB lehet (20 bit), de a deszkriptor egyik bitje meghatározza a szegmens felbontását is (granularity), vagyis, hogy ez a méret byte-ban vagy 4 KB-os lapokban értendő. Az utóbbi esetben a szegmens mérete maximálisan 4 GB lehet.

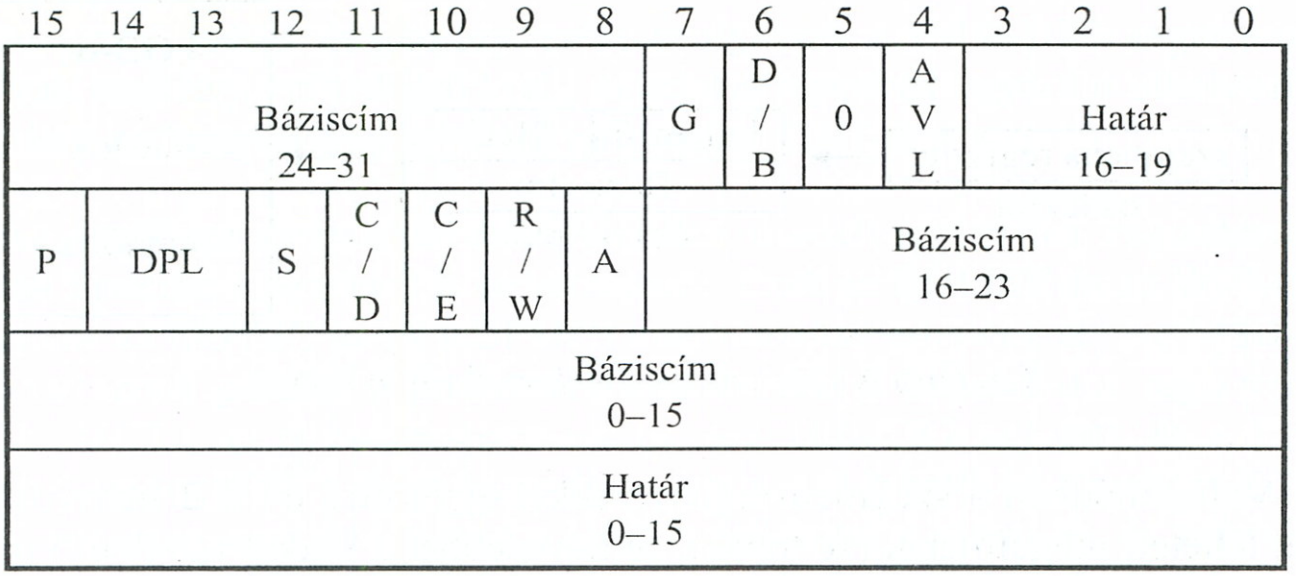
Ha esetleg valakinek a 4 GB-os szegmenshatár mégis kevésnek tűnne, az megnyugodhat, hiszen az újabb processzorok kiterjesztéseinek használatával lapozás esetén a lapok mérete 4 MB-ra is állítható, és elérhető, hogy a megcímezhető memória mérete akár 64 GB legyen.

A lapozás engedélyezésével virtuális címek is használhatóak, a 32 bites virtuális címet a lapozó egység alakítja át tényleges memóriacímekké. Erről a címátalakításról a lapozással foglalkozó fejezetben lesz szó.

1.4. A deszkriptorok

A deszkriptoroknak két fő típusa van: a szegmens-, és a rendszerdeszkriptorok. A memóriacímzéshez csak a szegmensdeszkriptorok használhatóak, a rendszerdeszkriptorok pedig bizonyos objektumokra, például kapukra, taszkokra, lokális deszkriptortáblázatokra (LDT) hivatkoznak. Minden deszkriptor 8 byte-os, és ezek táblázatokban vannak összegyűjtve.

A szegmensdeszkriptorok határozzák meg az egyes szegmensek báziscímét (hol kezdődnek a memóriában), határát (méretét), és a jellemzőit. A deszkriptor típusát és elérését meghatározó bitek az 1.6. ábrán szerepelnek.



1.5. ábra A szegmensdeszkriptorok felépítése

A bit neve	A bit jelentése
G	Felbontás (granularity) bit; ez határozza meg, hogy a szegmens mérete byte-ban, vagy 4 KB-os lapokban értendő.
D/B	A kódszegmensek esetén a D bit határozza meg az operandusok alapértelmezett (Default) méretét (16 vagy 32 bit), a B (Big) bit pedig az adatszegmensek méretét befolyásolja.
AVL	Ezt a bitet a rendszerprogramozók szabadon használhatják.
P	Ez a bit jelzi, hogy az adott szegmens a memóriában jelen van-e (Present), vagy nincs.
DPL	A deszkriptor privilégiumszintje (Descriptor Privilege Level)
S	A deszkriptor típusa; ha 0, akkor rendszer-, ha 1, akkor szegmensdeszkriptor
C/D	A szegmens típusa; ha 1, akkor kód-, ha 0, akkor adatszegmens. Ez a bit hatással van a következő két bit jelentésére.
C/E	Ha C = 1, akkor a kódszegmens illeszkedő (Conforming). Ha E = 1, akkor az adatszegmens (verem) lefelé bővülő (Expand down).
R/W	Ha R = 1, akkor a kódszegmens olvasható is, egyébként csak végrehajtható. Ha W = 1, akkor az adatszegmens írható is, egyébként csak olvasható.
A	Ezt a bitet a processzor automatikusan 1-re állítja, ha a deszkriptorra hivatkozás történt (Accessed).

1.6. ábra A szegmensdeszkriptorokban szereplő bitek

A deszkriptorokban a limit csak húsz bites, azaz egy szegmens mérete alapállapotban 1 MB lehet. Mivel gyakran szükség van 1 MB-nál nagyobb szegmensekre is, a G bit 1-re állításával a szegmens méretét a deszkriptorhatár 4096-szorosa adja. Ezzel a szegmensméret látszólag 32 bitessé válik, vagyis így akár 4 GB-os szegmensek (1 MB * 4 KB) is létrehozhatóak.

A D/B bit definiálja az adott szegmens méretét, vagyis hogy 16 vagy 32 bites szegmensről van-e szó. Ez a bit az utasítások végrehajtásakor játszik szerepet az operandusok és a címek méretének meghatározásánál.

A P bitnek elsősorban a virtuális memóriát használó rendszerekben van jelentősége, mivel ezzel jelezhető, hogy a szegmens a fizikai memóriában vagy a háttértáron van-e. Ennek a bitnek a felhasználásával a lapozás engedélyezése nélkül is megvalósítható a virtuális memória, ekkor teljes szegmenseket kell a memória és a háttértár között cserélgetni. Amikor egy olyan deszkriptorra történik hivatkozás, amelynek a P bitje 0-ra van állítva, akkor „szegmens nincs jelen" kivétel keletkezik. Ekkor az operációs rendszer – a taszk számára észrevétlenül – betöltheti az adott szegmenset a memóriába, majd visszaadhatja a vezérlést a taszknak.

A DPL bitek a deszkriptorok privilégiumszintjét határozzák meg, vagyis hogy az adott deszkriptort mely szintekről lehet elérni. A privilégiumszintekről egy későbbi fejezetben lesz szó.

Az S bit jelzi, hogy az adott deszkriptor egy szegmens- (S = 1), vagy egy rendszerdeszkriptor (S = 0). Az S bittől függően a típus mező bitjei más jelentést kapnak. A típusbitek segítségével sokféle deszkriptorfajta előállítható. A gyakorlatban legtöbbször előforduló deszkriptorok az alábbi táblázatokban szerepelnek.

C/D	C/E	R/W	Megnevezés
1	0	0	csak olvasható kódszegmens
1	0	1	végrehajtható és olvasható kódszegmens
0	0	0	írásvédett adatszegmens
0	0	1	írható és olvasható adatszegmens
0	1	1	lefelé bővülő veremszegmens

1.7. ábra A szegmensdeszkriptorok típusai

Típus	Megnevezés
2	Lokális deszkriptortáblázat (LDT)
5	Taszkkapu
9	Taszk-állapotszegmens (TSS)
12	Call-kapu
14	Megszakításkapu
15	Trapkapu

1.8. ábra A rendszerdeszkriptorok típusai

Szegmensdeszkriptorok esetén a típust az A bit értéke nem befolyásolja, mivel ezt a bitet a processzor automatikusan állítja be, tehát ennek változásával nem jön létre új deszkriptortípus.

1.5. A deszkriptortáblázatok

Háromféle deszkriptortáblázat létezik: a globális (GDT), a lokális (LDT) és a megszakítás-deszkriptortáblázat (IDT). Ezeknek a táblázatoknak a báziscímét (32 bit) és méretét (16 bit) három darab 6 byte-os regiszter tartalmazza, a GDTR, az LDTR és az IDTR.

A GDTR mutat a GDT-re, amely maximum 8191 deszkriptort tartalmazhat (a 0. bejegyzés használaton kívül van). Az LDT információit az LDTR tartalmazza, amelyen belül 8192 deszkriptor lehet (a 0. is használható), és ezeket csak bizonyos taszkok érhetik el. Az IDTR pedig a megszakítás-táblázat kezdetét és méretét jelzi, amelyben az egyes megszakítás-deszkriptorok (kapuk) szerepelnek.

A fenti három regiszterbe új érték betöltése értelemszerűen a 6 byte-os memóriaoperandusra (qword) hivatkozó LGDT, LLDT, LIDT utasításokkal lehetséges, a regiszterek tartalmának tárolására pedig az SGDT, az SLDT és az SIDT utasítások szolgálnak.

Olyan védett módú programok esetén, amelyek csak néhány globális deszkriptort használnak, a GDT mérete lehet kisebb a maximális 64 KB-nál, az LDT akár teljesen el is hagyható, mivel lokális deszkriptorokat főleg multitaszking rendszerekben alkalmaznak. Előfordulhat, hogy minden

taszknak külön LDT-je van, de használhat akár több taszk közös LDT-t is. Ha nincs kihasználva mind a 256 megszakítás, akkor az IDT méretét is lehet az igényeknek megfelelően alakítani.

1.6. A szelektorok

Védett módban a szegmensregiszterek a báziscím helyett szelektorokat tartalmaznak. A szelektorok felső 13 bitje egy indexet alkot, amely egy deszkriptor sorszámát adja meg a TI (Table Indicator) bit által jelzett táblázaton belül (0 = GDT, 1 = LDT).

1.9. ábra A szelektorok formátuma

A szelektor alsó két bitje az igényelt privilegizálási szint (Requested Privilege Level), amellyel megakadályozható, hogy egy alacsony privilégiumszintű program egy felsőbb szintű programon keresztül elvégeztessen olyan műveletet, amelyhez nincs joga, és amelyre önmagában nem is lenne képes. Például egy felhasználói program a kernel vagy az eszközmeghajtók szegmenseit nem érheti el közvetlenül, de megpróbálhatja azokat lemezre kimenteni, esetleg felülírni, mivel a lemezt kezelő eljárások már megfelelő jogosultságokkal rendelkeznek ehhez. A szelektorokban lévő RPL mező használatával ezek a kísérletek viszont kiszűrhetőek, és a védelmi mechanizmuson keresztül megakadályozhatóak.

1.7. Memóriacímzés egy példán keresztül

Miután megismerkedtünk a memóriacímzés három fő alkotóelemével, a szelektorokkal, a deszkriptorokkal és a táblázatokkal, nézzük meg, hogy működik ez a gyakorlatban.

Védett módban a szegmensregiszterekbe tehát nem báziscímet, hanem egy szelektort töltünk be, amely három részből áll: egy indexből, és a TI, RPL mezőkből. A TI bit által jelzett táblázatból (globális vagy lokális) az index által meghatározott sorszámú deszkriptor betöltődik az adott szeg-

mensregiszter „láthatatlan" részébe (cache). Ezután bármikor hivatkozunk az adott szegmensregiszterrel egy memóriacímre, az ofszet címéhez hozzáadódik a deszkriptorban lévő báziscím, és ezek összegéből keletkezik a 32 bites lineáris cím.

Vizsgáljuk meg, hogy mit eredményez például a következő néhány sor:

```
mov   eax,10h
mov   ds,ax
mov   [eax],eax
```

Bontsuk fel alkotórészeire a 10h (00010000b) szelektort: az index 2, a TI a globális deszkriptortáblázatra mutat, az RPL-nek pedig nincs szerepe a cím képzésében. Tehát a szelektor a GDT második deszkriptorára mutat.

	Bázis 31 24	**Jellemzők**		**Bázis** 23		0	**Határ** 15	0
0								
1								
2	00	40	92	10	00	00	F0	00
3								

1.10. ábra Egy képzeletbeli GDT

A fenti ábrán lévő 32 bites adatszegmens deszkriptor báziscíméhez (100000h) hozzá kell adni az utasításban szereplő ofszetcímet (10h), és ezzel megkaptuk a teljes címet, vagyis az utasítással az 1 MB utáni 10h címen lévő duplaszóba 10h-t rakunk.

Nézzünk meg egy kicsit összetettebb példát is:

```
mov   ax,7Fh
mov   fs,ax
mov   eax,fs:[eax+eax*8+00001220h]
```

Először az FS-be töltött szelektort az előzőhöz hasonló módon felbontjuk:

Index													TI	RPL	
0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1

1.11. ábra A 7Fh szelektor

Ez tehát egy 3. szintű lokális deszkriptor (TI = 1), amely az LDT 15. pozíciójában (7Fh/8) található.

	Bázis 31 24	Jellemzők		Bázis 23		0	Határ 15	0
13								
14								
15	40	40	F0	00	40	00	80	00
16								

1.12. ábra Egy képzeletbeli LDT

A 15. lokális deszkriptor a jellemzők (F0h) alapján egy jelenlévő (P = 1), 3. privilégiumszintű (DPL = 3), csak olvasható adatszegmens. Ennek báziscíme 40004000h, vagyis a szegmens egy kicsivel az 1 GB felett kezdődik, és 32 KB méretű (8000h). Ehhez a báziscímhez hozzá kell adni az EAX*9 értéket (477h), illetve az utasításban szereplő 1220h-t. Ezek alapján a fenti néhány utasítás betölti az EAX regiszterbe a 40005697h lineáris címen lévő adatot.

A védett módú memóriacímzés alappillérei tehát a szelektorok, a deszkriptorok és a deszkriptortáblázatok. Ezek működése első látásra nehéznek tűnhet a valós módhoz képest (ahol csak 16-tal kellett szorozni), viszont a védett módú címzésmód sokkal több lehetőséget rejt magában.

1.8. A 16 és 32 bites szegmensek

A szegmensdeszkriptorokban lévő D és B bitek határozzák meg a legtöbb utasításnál az operandusok és a címek alapértelmezett méretét. Ha ez a bit 0, akkor az adott szegmens 16 bites, ha a bit 1, akkor 32 bites.

Természetesen egy 16 bites kódszegmensben is lehet használni 32 bites operandusokat, és fordítva. A deszkriptorok alapértelmezése úgy változtatható meg, ha az utasítások előtt operandusméret-prefixet (66h) és címméret-prefixet (67h) helyezünk el.

A korábbi 16 bites processzorokon az utasítások operandusmérete 8 és 16 bit volt, a kétféle állapotot a 0. bit jelezte az utasítások kódjában (0 = byte, 1 = szó). A 32 bites processzorok (illetve a 32 bites regiszterek és

memóriaoperandusok) megjelenésével nem hoztak létre új utasításkódokat (nem is igazán tudtak volna), hanem megpróbálták a régi 1 biten a 3 különböző operandusméretet megvalósítani. Ezért vezették be a prefixeket és a D/B bitet.

Az érintett utasítások legalsó bitje 0 esetén továbbra is byte operandust jelöl a szegmensek méretétől függetlenül. Ha a 0. bit 1, akkor az operandus lehet szó vagy duplaszó méretű is, ezt az alapértelmezés és a prefixek döntik el.

A következő példán jól megfigyelhető a 32 és a 16 bites operandusok és címek közötti kapcsolat. A szegmensek alapértelmezését a Use32 és a Use16 használatával jelezhetjük, a prefixeket majd a fordító helyezi el automatikusan. A MOV AL,mem utasítás kódja A0h, a 0. bit 1-re állításával pedig megkaphatjuk a MOV EAX,mem, illetve a MOV AX,mem kódját is. Ezek csak 1 byte-os utasítások (plusz a 2 vagy 4 byte-os ofszetcím), a többi általános regiszter esetén a MOV reg,mem viszont már 2 byte-os lenne.

A példában szerepel minden lehetséges variáció: a 16 és 32 bites kódszegmensekből hivatkozunk a 16 és 32 bites adatszegmensekben lévő byte, szó és duplaszó méretű adatokra.

```
Data32 Segment Use32 'Data'
D32    dd 0
W32    dw 0
B32    db 0
Data32 Ends

Data16 Segment Use16 'Data'
D16    dd 0
W16    dw 0
B16    db 0
Data16 Ends

Code32 Segment Use32 'Code'

       mov  eax,D32          ; A1 00000000
       mov  eax,D16          ; 67 A1 0000
       mov  ax,W32           ; 66 A1 00000004
       mov  ax,W16           ; 66 67 A1 0004
       mov  al,B32           ; A0 00000006
       mov  al,B16           ; 67 A0 0006

Code32 Ends

Code16 Segment Use16 'Code'
```

```
        mov   eax,D32           ; 66 67 A1 00000000
        mov   eax,D16           ; 66 A1 0000
        mov   ax,W32            ; 67 A1 00000004
        mov   ax,W16            ; A1 0004
        mov   al,B32            ; 67 A0 0006
        mov   al,B16            ; A0 00000006

Code16 Ends
```

Tehát ha a kódszegmens, az adatszegmens és az adat típusa azonos, akkor nincs szükség prefixekre. Ha az adat és a kódszegmens típusa eltérő, akkor megjelenik a 66h prefix, ha a kódszegmens és az adatszegmens típusa eltérő, akkor pedig a 67h prefix, és az operandus címe 2 (vagy 4) byte-osra változik. Ha az operandus és az adatszegmens típusa is más, mint a kódszegmensé, akkor mindkét prefix szerepel.

A 16 és 32 bites szegmensek közötti különbség nemcsak a kód-, és adatszegmensek esetén jelentkezik, hanem a verem használatakor is, ezért a B bitet a veremszegmensek deszkriptorában is helyesen kell beállítani. (Különben például a PUSH EAX utasítás PUSH AX-ként értelmeződik, és a 4 byte helyett 2 byte-ot helyez el a veremben.)

Mint látható, arra vigyázni kell, hogy a védett módú programok futásakor az egyes szegmensek típusa úgy legyen beállítva, mint ahogy a fordító feltételezte, vagyis egy 16 bites program például ne egy 32 bites kódszegmensben fusson.

A prefixek egyéb utasításokra is hatással vannak, mint például az IRET-re. Az IRET 16 bites változata a verem tetején lévő 6 byte-ot tölti be a CS:IP és FLAGS regiszterekbe, míg a 32 bites változat (IRETD) a felső 12 byte-ot (3 duplaszót). Egy 16 bites szegmensben (és valós módban) az IRET kódja CFh, az IRETD kódja 66h és CFh, míg egy 32 bites szegmensben fordítva van, az IRETD kódja csak a CFh.

Az LGDT és az LIDT utasítások működését is befolyásolja az operandusméret. Elvileg mindkét utasítás egy 6 byte-os adatterületről betölti a 16 bites határt és a 32 bites báziscímet a megfelelő regiszterbe. A 16 bites verziók viszont a báziscímnek csak az alsó 24 bitjét tárolják. Emiatt például a valós módban végrehajtott LGDT utasítással a globális deszkriptortáblázat csak a 16 MB alatti területen helyezhető el. Természetesen, ha a 66h prefix szerepel az LGDT előtt, akkor a teljes 32 bites báziscím bekerül a GDTR-be, vagyis így a GDT már a 16 MB felett is elhelyezkedhet.

2. Védelem a védett módban

A védelmi mechanizmus célja az, hogy segítsen felismerni az egyes programhibákat, valamint lehetővé tegye a biztonságos és megbízható operációs rendszerek fejlesztését. Sajnos, a legtöbb operációs rendszer csak kis mértékben használja ki a processzor által kínált védelmi lehetőségeket, ezért ezek általában megbízhatatlanok, és nagy biztonsági lyukakat tartalmaznak. Néha a biztonságot is csak az óriási méret és a megnövekedett erőforrásigény árán sikerül megvalósítaniuk.

Védett módban sok olyan új elem van, amelyek segítségével roppant hatásos védelemi rendszer alakítható ki. Míg valós módban lényegében minden program korlátozás nélkül megtehetett mindent, védett módban már megakadályozható, hogy egyszerű felhasználói programok kárt okozzanak egymásban, a rendszerben, vagy a perifériákban.

A védelem több szintből tevődik össze:

Típusellenőrzés	A programok hogyan férhetnek hozzá a különböző típusú szegmensekhez
Határellenőrzés	A memóriahivatkozások csak a szegmensek határain belül történjenek
Privilégiumszintek	Az egyes taszkoknak melyik privilégiumszintű szegmensekhez van hozzáférése
Privilegizált utasítások	A privilegizált és I/O utasítások használatának korlátozása
Lapozás	Lapszintű védelem, a lapozás engedélyezésével a biztonság is fokozható

2.1. Típusellenőrzés

A típusellenőrzés biztosítja, hogy a szegmenseket a rendeltetésüknek megfelelően használják fel. Ez az ellenőrzés mindig végrehajtódik, ha például egy szegmensregiszterbe új értéket töltünk be, vagy ha egy memóriacímre hivatkozunk. Ha próbálkozás történik egy helytelen művelet végre-

hajtására, akkor egy kivétel keletkezik. (A kivételekről egy későbbi fejezetben lesz szó.)

Ilyen eset például, ha a CS-be egy adatszegmens szelektorát töltjük be, vagy fordítva, ha a DS-be egy csak végrehajtható kódszegmens szelektorát, illetve, ha megpróbálunk írni egy kódszegmensbe vagy egy csak olvasható adatszegmensbe. Védett módban a kódszegmensbe nem lehet írni, és olvasni is csak olyanból lehet, amelyik deszkriptorában a Read bit 1-re van állítva, továbbá az SS-be is kizárólag írható és olvasható adatszegmens szelektora tölthető be.

Ha szükség van arra, hogy a kódszegmenset módosítsuk, akkor létre kell hozni egy olyan írható és olvasható adatszegmenset, amelynek báziscíme megegyezik a kódszegmensével, és ekkor az adatszegmenset nyugodtan lehet változtatni. (Közvetlenül a módosítás után szükség lehet egy ugró utasításra is, mert elképzelhető, hogy az eredeti kód már a cache-ben van, és az hajtódik végre, nem pedig az új.)

2.2. Határellenőrzés

A határellenőrzés elve nagyon egyszerű: megakadályozza a programokat abban, hogy a szegmensek határain kívüli területet érjenek el.

Valós módban minden szegmens mérete kötelezően 64 KB volt. Védett módban viszont a deszkriptorban lévő 20 bites határ segítségével megadható a szegmens pontos mérete is. Ezáltal csökkenthető annak az esélye, hogy egy program olyan adatokhoz fér hozzá, vagy ír felül, amely már nem tartozik hozzá. Ha például védett módban van egy 100 byte-os adatszegmens, akkor a 100 byte-n túli területek elérése (írása, olvasása) már nem jár sikerrel.

Lefelé bővülő adatszegmensek (verem) esetén a határ másképp van értelmezve. Ekkor csak a határ és a maximum közötti hivatkozások érvényesek, nem pedig a 0 és a határ közöttiek. Tegyük fel, hogy a veremszegmens határa 0FFFFF800h, vagyis a verem mérete 2 KB, és a deszkriptorban a B bit értéke 1. A verembe mindaddig helyezhető el új adat (PUSH-al), amíg az ESP a határ fölé mutat, de amint eléri a 0FFFFF800h értéket, veremhiba generálódik. Ekkor az operációs rendszernek lehetősége van arra, hogy a taszk

számára észrevétlenül megnövelje a verem méretét például 4 KB-ra, vagyis a határt 0FFFFF000h-ra állítsa, és visszaadja a vezérlést a taszknak. Ez csak a lefelé bővülő adatszegmensekkel érhető el, mert különben a verem alját akkor érjük el, amikor az ESP értéke 0, és ekkor a verem mérete nem növelhető tovább.

2.3. Privilégiumszintek

Nem lenne igazán védett a védett mód, ha nem lehetne pontosan meghatározni, hogy az egyes programok milyen jogosultságokkal rendelkezzenek, illetve mihez férhetnek hozzá: mely szegmensekhez, memóriaterületekhez, rendszerobjektumokhoz; elérhetik-e I/O műveleteken keresztül a hardvereszközöket, stb. Ez a védelmi rendszeren belül a privilégiumszintekkel valósítható meg. A privilégiumszintekkel való visszaélés vagy véletlen hiba könnyen észrevehető, a védelmi szabályok legkisebb megsértése is kivételt okoz.

Négy privilégiumszint van 0-tól 3-ig számozva. A 0. szinten futó rutinok rendelkeznek a legtöbb jogosultsággal, a 3. szintűek pedig a legkevesebbel. Egy védett módú rendszer megtervezésekor több lehetőség közül lehet választani a privilégiumszintek kialakításánál, vagyis hogy a négy szint közül mennyi legyen kihasználva.

Megoldható akár az is, hogy egy egyszerű védett módú program csak a 0. szintet használja. Ez viszont egyáltalán nem biztonságos megoldás, hiszen a legkisebb programhiba is – a valós módhoz hasonlóan – komoly problémákhoz vezethet (pl. lefagyás).

Egy fokkal biztonságosabb, ha a 0. és a 3. szintet is használjuk. Ekkor a jogosultságokkal rendelkező rutinok (kernel, eszközmeghajtók, stb.) a 0. szinten futnak, az alkalmazói programok pedig a 3. szinten.

Ennél még sokkal biztonságosabbá tehető a rendszer, ha a többi privilégiumszintet is kihasználjuk. A 0. szinten csak a legalapvetőbb, kulcsfontosságú rutinok (kivételkezelők, rendszerhívások, erőforráskezelés, ütemezés, lapozás, stb.) találhatóak, az első szinten helyezkednek el a hardvereszközöket kezelő driverek, esetleg a második szinten a parancsértelmezők vannak, míg a harmadikon az alkalmazói programok futnak.

2.3.1. A privilégiumszintek ellenőrzése

A privilégiumszintek ellenőrzését a következők befolyásolják:

CPL	Az aktuális privilégiumszintet (Current Privilege Level) a CS regiszterben lévő szelektor legalsó két bitje határozza meg
DPL	A deszkriptorok privilégiumszintjét (Descriptor Privilege Level) a DPL bitek adják meg
RPL	Az igényelt privilegizálási szintet (Requested Privilege Level) a szelektorok legalsó két bitje határozza meg

Ezt a három elemet a processzor mindig ellenőrzi, például ha egy szegmensregiszterbe új szelektort töltünk be. A privilégiumszint ellenőrzés különböző módon történik az adat-, a verem-, és a kódszegmensek esetén.

2.3.2. A privilégiumszintek – Az adatok

Az aktuálisan futó kód elérheti az azonos vagy alacsonyabb (számszerint magasabb) privilégiumszintű adatszegmenseket (ha DPL >= CPL és DPL >= RPL), a magasabbakat viszont nem. Ezzel többek között elkerülhető az is, hogy egy alkalmazói program kárt tegyen a kernel adatszegmensében.

A szintekre osztás segítségével az adatrejtés is megvalósítható, tehát a több jogosultsággal rendelkező rétegek elfedhetik az alacsonyabb rétegek elől a privát adatokat, stb. Míg például valós módban a BIOS rendszerváltozói (0040:0000h-tól) szabadon módosíthatóak voltak, ugyanez egy védett módú operációs rendszernél már megakadályozható.

Egy adatszegmens regiszter akár egy olyan kódszegmens szelektorral is feltölthető, amely olvasható is. Ezáltal a kódszegmensben tárolt adatok is elérhetőek. Ezt a módszert használja például a legtöbb megszakítás-kezelő is ahhoz, hogy a szegmensregiszterekbe a szükséges értékeket betöltse. A megszakításba való belépéskor ugyanis a DS, az ES, az FS, és a GS bármit tartalmazhat, és a kezelő adatszegmens szelektora nem mindig állandó érték. A kódszegmensben elhelyezett szelektorok a CS regiszter segítségével már elérhetőek, és a többi szegmensregiszter is könnyen feltölthető a megfelelő értékekkel.

2.3.3. A privilégiumszintek – A verem

Védett módban egy taszkon belül minden privilégiumszinthez külön verem tartozik. A legegyszerűbb esetben, amikor csak a 0. szintet használjuk, csak egy veremre van szükség, viszont egy három privilégiumszintet használó operációs rendszer esetén már minden egyes taszkhoz három különböző vermet kell biztosítani. Ha az alkalmazói program egy magasabb privilégiumszinten lévő rutint (például a kernelt vagy egy drivert) hív meg, akkor a processzor automatikusan egy „biztonságosabb" veremre kapcsol át. A különböző szinten lévő vermek adatait a taszk-állapotszegmens (TSS) tartalmazza. Ez a megoldás azért célszerű, mert az alkalmazói program legfeljebb a saját vermét teheti tönkre, de a magasabb szintű vermeket nem érheti el.

Szükség lehet arra is, hogy magasabb privilégiumszintű eljárásoknak a vermen keresztül adjunk át paramétereket. A call-kapuk deszkriptorában megadható a paraméterek száma, ami a régi veremből automatikusan az újba másolódik.

Amikor a verem szegmensregiszterét (SS) feltöltjük, a szelektornak az aktuális privilégiumszinthez kell igazodnia (RPL = CPL = DPL).

2.3.4. A privilégiumszintek – A kódszegmensek

A vezérlés átadása egy másik kódszegmensre történhet a FAR CALL, FAR JMP, RETF, INT vagy az IRET utasításokkal. A vezérlés átadása során a cél szelektora a következő deszkriptorokra mutathat:

- egy másik kódszegmens
- call-kapu, megszakítás- vagy trapkapu
- taszk-állapotszegmens vagy taszkkapu

A privilégiumszint ellenőrzése különböző módon történhet a cél típusától és a használt utasítástól függően. Az alábbiakban a CALL, JMP és RET utasítások segítségével és a call-kapun keresztül történő vezérlésátadásról lesz szó, a taszkkapcsolás ismertetésére pedig a multitaszking fejezetben kerül sor.

2.3.5. Vezérlésátadás másik kódszegmensre

Amikor egy másik kódszegmensre adjuk át a vezérlést a CALL vagy a JMP utasítások segítségével, két eset lehetséges attól függően, hogy a cél kódszegmens típusa illeszkedő (conforming), vagy nem (non-conforming).

A gyakorlatban a legtöbb kódszegmens nem illeszkedő, ekkor a hívó és a hívott privilégiumszintnek meg kell egyeznie (CPL = DPL), különben kivétel keletkezik.

Az illeszkedő szegmensek akkor használatosak, amikor egy kódszegmenset egyszerre több privilégiumszintről is elérhetővé szeretnénk tenni. Ha vezérlésátadás történik egy illeszkedő szegmensre, a CPL a hívó privilégiumszintjével lesz azonos, vagyis a processzor nem kapcsol magasabb szintre. Az illeszkedő kódszegmens DPL-je jelzi azt a privilégiumszintet, ahonnan ez a szegmens még meghívható. Ha az aktuális privilégiumszint számszerűen kisebb, mint a deszkriptor privilégiumszintje (CPL < DPL), akkor kivétel keletkezik.

Például ha egy illeszkedő kódszegmens DPL-je 1, akkor az ebben lévő eljárásokat az 1., a 2. és a 3. privilégiumszintekről meghívhatják (a CPL az eredeti marad), viszont a (legprivilegizáltabb) 0. szintről már nem. Ezáltal megakadályozható, hogy a felügyelői szintről olyan eljárásokat hívjunk meg, amelyek nem biztonságosak, és veszélyeztetnék az egész rendszer működését.

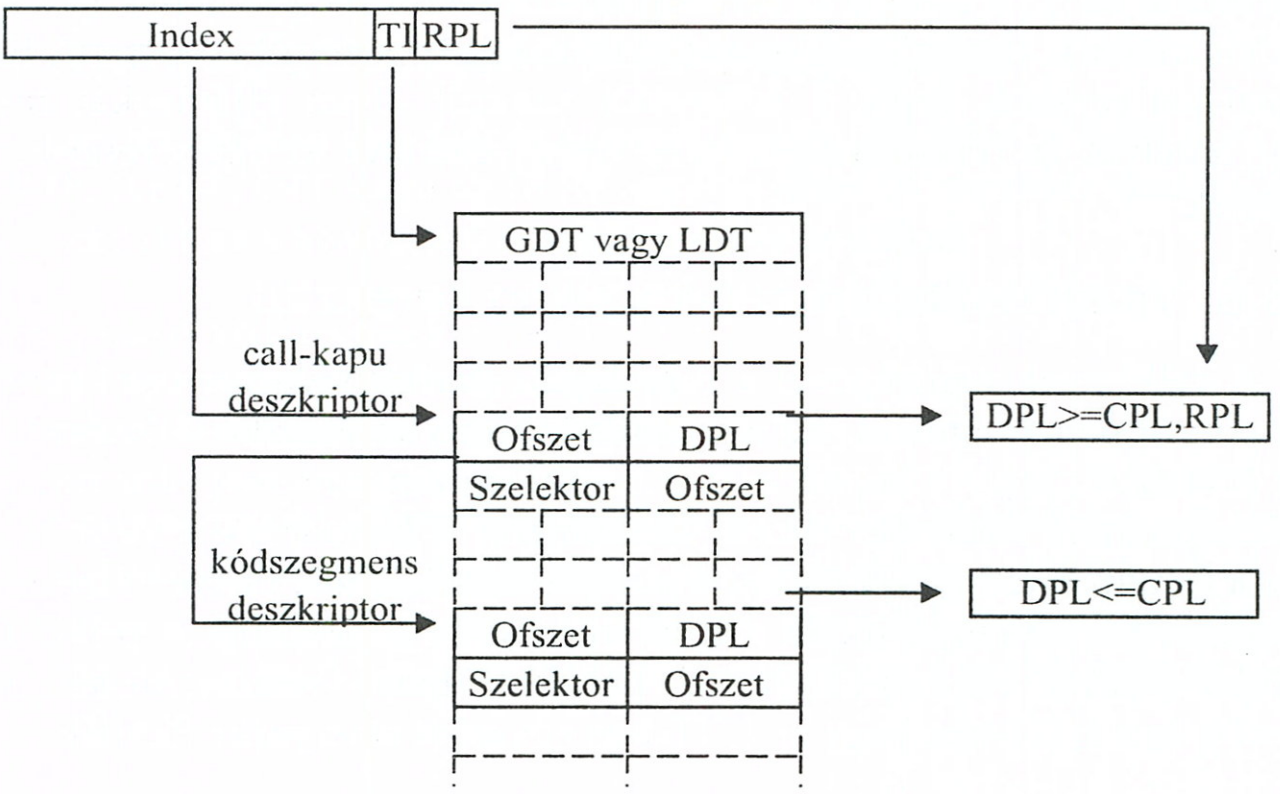
2.3.6. Vezérlésátadás call-kapukon keresztül

A call-kapuk segítségével a végrehajtás átadható alacsonyabb privilégiumszintről magasabbra is. Ez abban különbözik az illeszkedő kódszegmensek hívásától, hogy a kapuk esetén a privilégiumszint megváltozik, míg a kódszegmensek közvetlen hívása esetén nem. A kapudeszkriptorok négy fő mezőt tartalmaznak: a kapu belépési pontjának szelektorát, illetve ofszetcímét, a veremből átmásolandó dupla szavak számát és a deszkriptor privilégiumszintjét (DPL). A call-kapuk 12 (1100b) típusú rendszerdeszkriptorok (S = 0), és a GDT-ben vagy az LDT-ben egyaránt elhelyezkedhetnek.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Belépési pont ofszetcíme 16–31															
P	DPL		S = 0	Típus 1 1 0 0				0	0	0	Paraméterek száma				
Belépési pont szelektora 0–15															
Belépési pont ofszetcíme 0–15															

2.1. ábra A call-kapu deszkriptorok felépítése

A deszkriptorban lévő paraméterszámlálóval megadható a dupla (4 byte-os) szavak száma, amely jelzi a verem átkapcsolásakor átmásolandó paraméterek számát. Ezáltal a call-kapuk szinte ugyanúgy használhatóak, ahogyan a közeli eljárás hívások, vagyis paraméterek adhatók át a vermen keresztül, és ezeket a verem átváltása ellenére is eléri a hívott eljárás. Ez a mező 5 bites, ami 63 duplaszó átadását engedi meg.



2.2. ábra Privilégiumszint-ellenőrzés vázlatosan a call-kapuk esetén

A call-kapuk DPL mezőjének működése a szokásos, vagyis ezek a bitek határozzák meg, hogy mely privilégiumszinten lévő programok használhatják a kaput. Például egy DPL = 2 kaput meghívhatnak a 0., az 1. és a 2. szinten futó programok, ha viszont egy 3. szintű kód próbálkozik ugyanezzel, akkor ez meghiúsul (kivétel keletkezik). Előfordulhat, hogy a CPL nem egyezik meg a szelektorban lévő RPL szintjével, ekkor a processzor a kettő közül a magasabb értéket veszi figyelembe. A kódszegmens deszkriptor (amelyre a kapu mutat) DPL-jének ugyancsak nagyobb vagy egyenlőnek kell lennie, mint az aktuális privilégiumszint, különben általános védelmi hiba keletkezik.

2.3.7. A RETF utasítás

Egy távoli eljárásból, amit a FAR CALL-al hívtunk meg, a RETF utasítás használatával lehet visszatérni. Ekkor is szükség van a privilégiumszintek ellenőrzésére, mivel a veremben lévő visszatérési cím könnyedén módosítható.

A RETF utasítással egy azonos vagy alacsonyabban privilégiumszintű szegmensre adható át a vezérlés. Ha a veremben lévő CS regiszter RPL-je alacsonyabb szintű, mint a CPL, akkor a visszatérés során a hívó privilégiumszintjére kapcsolunk vissza, ha pedig azonos szintűek, egyszerűen csak egy másik kódszegmensre adódik át a vezérlés, anélkül, hogy a privilégiumszint megváltozna. Viszont kivétel keletkezik akkor, ha az igényelt privilégiumszint magasabb az aktuálisnál (ha RPL > CPL).

2.4. Privilegizált és érzékeny utasítások

A védelmi mechanizmus megakadályozza az operációs rendszer számára kulcsfontosságú utasítások végrehajtását is a kevésbé privilegizált szinteken. Néhány utasítás (2.3. ábra) végrehajtása korlátozva van csak a 0. privilégiumszintű (felügyelői) taszkok számára, míg mások az I/O privilégiumszintre (IOPL) érzékenyek.

LGDT	LMSW
LLDT	MOV CRn
LIDT	MOV DRn
LTR	INVD
CLTS	INVLPG
HLT	WBINVD

2.3. ábra A privilegizált utasítások

Az EFLAGS 12. és 13. bitjei definiálják a taszkok I/O privilégiumszintjét, vagyis hogy mely szinteken végezhetnek I/O műveleteket.

12	13	IOPL szint
0	0	0
0	1	1
1	0	2
1	1	3

2.4. ábra Az IOPL értékek

Az I/O utasítások a külső hardvereszközöket kezelik a portokon keresztül (IN, INS, OUT, OUTS), illetve a megszakítások engedélyezésével és tiltásával (STI, CLI). Ha a processzor virtuális 8086-os üzemmódban van, akkor a 8086-os környezet emulálása érdekében a PUSHF, POPF, INT és az IRET is érzékeny (sensitive) utasításnak minősül.

Az IOPL határozza meg, hogy mely szinteken lehet ezeket az érzékeny utasításokat végrehajtani. Az IOPL értékét csak 0. szinten futó kód változtathatja meg, más szintről nem módosítható. Mivel minden taszknak saját EFLAGS regisztere van, ezért minden egyes taszknak külön beállítható az IOPL szintje. A taszk-állapotszegmensben lévő I/O engedélyező térképpel tovább finomítható az egyes portok elérése.

Az IOPL szintre érzékeny utasítások segítségével egy operációs rendszer teljes mértékben felügyelheti a külső eszközökhöz való hozzáférést. Ha például egy operációs rendszer alatt minden program a 3. szinten fut, és az IOPL értéke 0, akkor a programoknak a hardvereszközök eléréséhez az OS 0. szinten futó szolgáltatásait kell igénybe venniük. Az IOPL 3-ra állításával

(akár csak néhány taszk esetén) az alkalmazói programok közvetlenül is elérhetik a hardvereszközöket.

Ha az IOPL szintje magasabb, mint az aktuális taszk privilégiumszintje, akkor az érzékeny utasítások használatakor kivétel keletkezik. A kivételkezelő segítségével az operációs rendszer eldöntheti, hogy engedélyezi-e az adott utasítás végrehajtását, vagy megtagadja.

A memóriába leképezett (memory-mapped) I/O eszközök hozzáférése is szabályozható az operációs rendszer számára az adott memóriaterületre mutató deszkriptor, illetve szelektor létrehozásával, és a deszkriptor DPL-jének helyes beállításával.

3. A védett módú programozás magas szinten

3.1. Megszakítások és kivételek

Védett módban a megszakítás-mechanizmus hasonlóan működik, mint valós módban, kivéve hogy a 4 byte-os CS:IP címeket tartalmazó táblázatot felváltotta a 8 byte-os deszkriptorokból álló táblázat, az IDT.

Az IDT báziscímét és méretét az IDTR regiszter határozza meg, tehát már nem kell a 0000:0000 címen elhelyezkednie, és az sem kötelező, hogy a maximális 256 megszakítás-kezelő címét tartalmazza. Mivel a megszakítások véletlenszerűen (mind a négy szinten) is bekövetkezhetnek, a kapuk gondoskodnak arról, hogy az éppen futó taszk megszakadjon, és a vezérlés átadható legyen bármely privilégiumszintre az aktuális szinttől függetlenül.

Az IDT-ben három fajta rendszerdeszkriptor lehet: megszakításkapuk (típus = 14), trapkapuk (típus = 15) és taszkkapuk (típus = 5).

<table>
<tr><td colspan="16">15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</td></tr>
<tr><td colspan="16">Belépési pont ofszetcíme
16–31</td></tr>
<tr><td>P</td><td colspan="2">DPL</td><td>S
=
0</td><td colspan="4">Típus
1 1 1 0
1 1 1 1</td><td>0</td><td>0</td><td>0</td><td colspan="5">Fenntartva</td></tr>
<tr><td colspan="16">Belépési pont szelektora
0–15</td></tr>
<tr><td colspan="16">Belépési pont ofszetcíme
0–15</td></tr>
</table>

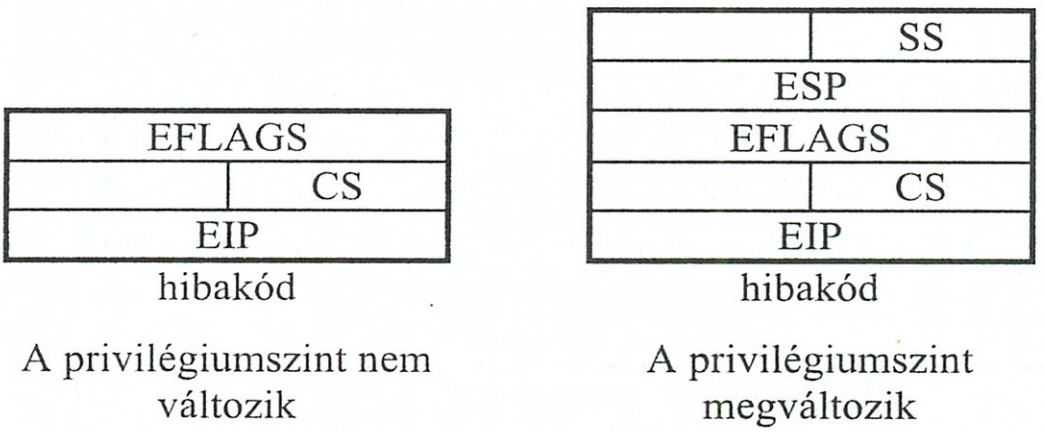
3.1. ábra A megszakításkapu és trapkapu deszkriptorok felépítése

A megszakításkapuk és a trap kapuk csak az EFLAGS regiszter megszakítás-engedélyező (Interrupt Flag) bitjének módosításában különböznek. A megszakításkapuk letiltják a megszakításokat (IF = 0), míg a trapkapuk

nem változtatják meg az IF értékét. Mivel a megszakítások hívásakor az EFLAGS elmentődik a verembe, az IRET végrehajtásakor az IF bit az eredeti értékére áll vissza.

A taszkkapukra hivatkozó megszakítások esetén taszkváltás történik a deszkriptorban szereplő TSS szelektor által jelzett taszkra. Ez akkor lehet hasznos, ha a megszakítás kezeléséhez egy teljesen külön környezetre van szükség. A megszakításkapukhoz képest kis időveszteség is keletkezik a taszkkapcsolás miatt, ezért a nagyon gyakori megszakításokat nem célszerű taszkkapun keresztül kezelni. Ugyancsak kerülni kell a taszkkapuk használatát azoknál a megszakításoknál, amelyek újra meghívódhatnak, mielőtt befejeződtek volna (re-entrant).

A megszakítások kiváltó oka lehet egy külső esemény (IRQ), egy kivétel (hiba) vagy egy utasítás (INT3, INT n, INTO és BOUND). Az IRQ és az utasítások által létrejött megszakítások esetén a veremben elhelyezett információ megegyezik (3 duplaszó: EFLAGS, CS és EIP), míg néhány kivételnél egy 32 bites hibakód is kerül a verem tetejére. Ha a privilégiumszint is megváltozik a megszakítás során, akkor a processzor a régi szint vermének a címét (SS:ESP) is elhelyezi veremben, és az SS:ESP regiszterekbe az új szinthez tartozó verem címe kerül.



3.2. ábra A verem tartalma egy megszakítás során

A megszakításból való kilépés a valós módhoz hasonlóan az IRET (pontosabban az IRETD) utasítással történik. Ekkor a processzor visszaállítja a korábbi állapotbiteket, majd a veremben lévő CS:EIP-re adja a vezérlést. Ha az aktuális privilegizálási szint (CPL) kisebb, mint a verembeli kódszegmens szelektorának RPL-je, akkor a privilégiumszint is megváltozik, és

az SS:ESP regiszterekbe a verem következő 2 duplaszava kerül. Ha az adott megszakítás vagy kivétel egy taszkkapun keresztül hívódott meg, akkor az NT bit 1-re változik. Ez esetben az IRETD utasítás végrehajtásakor a megszakított taszkra kapcsol vissza.

3.1.1. A megszakítások működésének bemutatása egy konkrét példán keresztül

Vizsgáljuk meg, mi történik például az INT 3 (opkód CCh) végrehajtásakor. A 3. megszakításvektor deszkriptorának fizikai címe az IDTR báziscímének és az adott vektor táblázatbeli ofszetcímének (3*8) összegéből adódik. Ez a deszkriptor tartalmazza a belépési pont szegmensének szelektorát és az ofszetcímét.

	Ofszet		Jellemzők		Szelektor		Ofszet	
	31	16					15	0
0								
1								
2								
3	00	00	9F	00	00	10	15	6A
4								

3.3. ábra Egy képzeletbeli IDT

A fenti ábrán tehát az INT 3 kezelőjének szelektora 10h, az ofszetcíme pedig 0000156Ah. A deszkriptor típusából (9Fh) kiderül, hogy ez egy jelenlévő trapkapu. A 10h szelektorból pedig megállapítható, hogy a megszakítás-kezelő kódszegmens deszkriptora a GDT második bejegyzésében található.

	Bázis		Jellemzők		Bázis			Határ	
	31	24			23		0	15	0
0									
1									
2	00		40	98	09	50	00	20	00
3									

3.4. ábra Egy képzeletbeli GDT

A fenti GDT részletben a kódszegmens báziscíme 95000h, ehhez kell hozzáadni az IDT 3. deszkriptorának ofszetcímét, hogy megkapjuk az INT 3 belépési pontját (95000h + 156Ah = 9656Ah).

3.1.2. Kivételek

A kivételek (exceptions) olyan speciális megszakítások, amelyek akkor keletkeznek, ha a processzor valamilyen hibát fedez fel egy utasítás végrehajtása közben. Ezeknek a kivételeknek egy része már valós módban illetve a korábbi processzorokon is jelentkezik, ilyen például az osztási hiba és az érvénytelen utasítás kivétel.

A védett módban ismert kivételek a 3.5. ábrán szerepelnek. Ezek három csoportra oszthatóak, vannak hibák (faults), trap-ek, és abortok. A hibák esetén a verembe mentett CS:EIP a hibát okozó utasításra mutat, így lehetőség van a hiba kijavítása után az adott utasítás újraindítására. A trap kivéte-

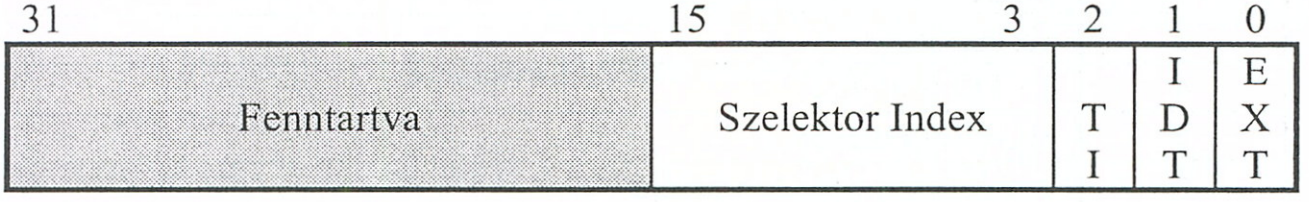
Sorszám	Megnevezés
0	osztási hiba
1	nyomkövető kivétel (debug)
2	NMI (Nonmaskable Interrupt)
3	töréspont
4	INTO utasítás
5	BOUND utasítás
6	ismeretlen utasítás kódja
7	eszköz nem elérhető
8	dupla hiba
9	koprocesszor szegmenshatár túllépés
10	érvénytelen TSS
11	szegmens nincs jelen
12	veremhiba
13	általános védelmi hiba
14	laphiba
15	fenntartva
16	koprocesszor hiba
17	illeszkedés ellenőrzés
18	géptípus ellenőrzés
19–31	fenntartva

3.5. ábra A kivételek listája

leknél (pl. a töréspont) az elmentett CS:EIP a következő utasításra mutat, akárcsak a szoftver megszakításoknál. Az abortok (pl. a duplahiba kivétel) okának kijavítása nehéz, és nem mindig lehetséges.

Néhány kivétel esetén a verem tetejére egy 32 bites hibakód is kerül. Ez a hibakód mindig a kivételkezelő vermébe kerül, még taszkkapcsolás esetén is, és a kivétel befejezése előtt ezt el kell távolítani a veremből, hogy folytatható legyen a megszakított utasítás (ha a hibát sikerült elhárítani). A 8. (dupla hiba) és a 17. (illeszkedés ellenőrzés) kivételek hibakódja mindig 0. A laphiba kivétel hibakódjának saját formátuma van, erről a lapozás című fejezetben lesz szó.

A 9–13. kivételek hibakódja a 3.6. ábrán látható. A hibakód alsó 16 bitje hasonlít a szelektorok felépítésére, de itt az RPL bitek helyett az EXT (1, ha a hibát külső esemény okozta) és az IDT (1, ha egy megszakításdeszkriptor a hiba oka) bitek szerepelnek. A TI (Table Indicator) bit jelentése változatlan, és azt jelzi, hogy egy globális (TI = 0) vagy egy lokális (TI = 1) deszkriptorral kapcsolatos-e a hiba. A leggyakoribb hibakódok a szegmensekkel kapcsolatosak, ekkor a hibakód egy RPL = 0 szelektornak felel meg.



3.6. ábra A hibakód általános formája

A 0. kivétel – az általános tévhittel ellentétben – nem csak akkor következik be, ha a DIV és az IDIV utasítás végrehajtásakor az osztó 0, hanem akkor is, ha az eredmény nem fér el a célregiszterben. Például az alábbi osztás is 0. kivételt okoz, mivel az 500h/5 = 100h már nem tárolható egy byte-on, az AL regiszterben:

```
mov  ax,500h
mov  bl,5
div  bl
```

Az osztási hiba elvileg kijavítható, ha sikerül felismerni a hiba forrását, vagyis hogy a kivételt 0-val való osztás, túlcsordulás vagy esetleg az INT 0 végrehajtása eredményezte. Mivel a veremben CS:EIP a hibát okozó utasí-

tásra mutat, ezért ez könnyen megállapítható, és az utasítás akár újra is indítható a hiba kijavítása után. Például a nullával való osztás esetén az operandusba (az osztó) 0 helyett 1-et töltünk, és ekkor az osztás már végrehajtható. Ennek persze semmi értelme, hiszen a kapott eredmény úgyis helytelen lesz, ezért egyszerűbb, ha például megszakítjuk az adott taszk futását.

Az 1. és a 3. kivétel a nyomkövetésben játszik szerepet. Mivel az INT 3 csak 1 byte-os utasítás, ezért remekül használható töréspontok létrehozására, de az INT 1 is alkalmas többféle töréspont (taszkkapcsolások, utasítás-, és adatcímek) megvalósítására. Ezzel a két kivétellel, illetve a nyomkövető regiszterek (DR0–DR7) felhasználásával eredményesen elvégezhető a hibák keresése és kijavítása, esetleg bizonyos programok működésének „kifürkészése".

Az NMI (NonMaskable Interrupt) hardverhibák esetén lép fel, és mint ahogy a neve jelzi, ez egy nem maszkolható megszakítás. Szokás szerint (az egyéb processzortípusokhoz hasonlóan) ha egy megszakításra azt mondják, hogy az nem maszkolható (nem tiltható le), akkor szinte biztos, hogy van valamilyen módszer a maszkolására. Ez esetben sincs másképpen, az NMI letiltása könnyen megvalósítható:

```
in  al,70h          ; olvasás a CMOS portról
or  al,10000000b    ; 7. bit: NMI letiltása
out 70h,al          ; visszaírás
in  al,70h          ; RTC zavar elkerülése
```

A 4. és az 5. kivételek működése a valós üzemmódhoz hasonló, vagyis aritmetikai túlcsordulás (INTO utasítás) és határsértés (BOUND utasítás) esetén keletkeznek. Ezeknek nincs igazán jelentősége, mivel közvetlen utasítások (pl. JO, JNO) használatával ugyanaz az eredmény sokkal kevesebb órajellel elérhető, és főleg akkor nagy az időveszteség, ha a kivételkezelő meghívásához más privilégiumszintre kell kapcsolni.

Ha az adott processzor egy számára ismeretlen vagy illegális operandusú utasítást próbál végrehajtani, akkor a 6. kivétel generálódik. Ugyanezt a kivételt okozza a busz lezárás (lock utasítás) helytelen használata is.

A 7., 9. és a 16. kivételek a koprocesszorral kapcsolatosak. A 7. kivétel (eszköz nem elérhető) például akkor keletkezik, ha a numerikus kopro-

cesszor nincs jelen a gépben, vagy ha a CR0 regiszter TS (Task Switched) bitje 1-re van állítva, ami jelzi, hogy taszkváltás történt. A kivétel segítségével az operációs rendszernek lehetősége van az FPU műveletek emulálására is, ha a koprocesszor nincs jelen.

Ha egy olyan kivétel generálódik, amelynek kezelője valamilyen oknál fogva nem indítható el (pl. veremhiba, érvénytelen TSS, szegmens nincs jelen), akkor dupla hiba akkor keletkezik. Ahhoz, hogy az eredeti kivétel újra elindítható legyen, meg kell szüntetni az első hiba okát. Ha ez a 8. kivétel sem hívódhat meg, akkor tripla hiba jön létre, és a processzor shutdown állapotba kerül. A dupla hiba okát általában nehezen lehet helyesen kijavítani, ezért vigyázni kell, hogy például az IDTR-ben lévő érték helyes legyen, és hogy a kivételkezelők végrehajtása során ne keletkezhessen egy újabb komoly kivétel.

A 10. kivétel (érvénytelen TSS) taszkváltások során keletkezhet, ha a régi, illetve az új TSS mérete kisebb, mint 103 byte, vagy ha az új TSS érvénytelen adatokat tartalmaz. Utóbbi eset akkor fordulhat elő, ha a szegmensregiszterekbe illegális szelektor töltődik be, ha az LDT érvénytelen, vagy ha a privilégiumszintekkel van probléma. Az érvénytelen TSS kivétel kezelőjét célszerű egy taszkkapun keresztül megvalósítani, hogy biztonságosan elhárítható legyen a hiba.

A 11. (szegmens nincs jelen) kivétel akkor jön létre, ha olyan deszkriptorra történik hivatkozás, amelyben a P bit 0-ra van állítva. E kivétel segítségével a lapozás engedélyezése nélkül is kialakítható a virtuális memória, ekkor nem az egyes lapokat kell mozgatni a memória és a háttértár között, hanem teljes szegmenseket, és a háttértárra mentett szegmensek deszkriptorában a P bitet 0-ra kell állítani (not present).

Veremhiba (12. kivétel) egyrészt határsértés esetén keletkezik, ha verem méretén túlra történik hivatkozás (például ha betelik a verem), másrészt ha az SS regiszterbe egy nem jelenlévő (P = 0) adatszegmensre mutató szelektor kerül. A 12. kivétel kezelőjét is külön taszkként kell megvalósítani, mivel ha a 0. privilégiumszinten történik veremhiba, akkor dupla hiba generálódik, és ha annak kezelője sem egy elkülönített taszk, akkor a hiba nem javítható ki.

Az általános védelmi hiba a leggyakrabban előforduló kivétel, és elsősorban a kevésbé biztonságos operációs rendszerekben, illetve a hibás programokban jelentkezik. Ez a kivétel akkor keletkezik, ha a védelmi mechanizmus bármilyen hibát fedez fel (például a védelem részben tárgyalt esetek valamelyikét).

A típusellenőrzés gondoskodik arról, hogy a szegmensregiszterekbe ne lehessen érvénytelen típusú deszkriptort betölteni vagy ne lehessen a kódszegmensbe írni, stb. Határsértést okoz, ha az adott szegmens határán túli területre vagy a deszkriptortáblázat méretén túli deszkriptorra történik hivatkozás. A privilégiumszintekkel, illetve az I/O utasításokkal való „visszaélés" is általános védelmi hibát eredményez.

Többek között 13. kivétel keletkezik a 15 byte-nál hosszabb utasítások végrehajtásakor, a GDT 0. deszkriptorával történő memóriahivatkozás, vagy egy foglalt taszkra való átkapcsolás esetén, illetve virtuális 8086-os módban egy megszakítás hívásakor (ha IOPL<3).

Ha a lapozás engedélyezett, akkor a laphiba (14. kivétel) is gyakran előfordulhat. Ez akkor jön létre, ha egy taszk megkísérel egy olyan lapot elérni, amely nincs jelen (P = 0), vagy amelyhez nincsen joga. Laphiba esetén a hibás lineáris cím a CR2 regiszterbe kerül, a veremben lévő hibakód pedig egyedi formátummal rendelkezik. Erről bővebben a lapozással foglalkozó fejezetben lesz szó.

Az illeszkedés és a processzortípus ellenőrzés kivétel alkalmazására ritkán kerül sor, ezek csak a 486-os (17. kivétel), illetve Pentium (18. kivétel), vagy jobb processzorokon fordulhatnak elő.

A kivételek segítségével tehát sokféle hiba felismerhető, és ezek bizonyos esetekben kijavíthatóak, továbbá sok egyéb funkció is megvalósítható. Szinte minden védett módú programban szükség van a kivételek valamilyen szintű kezelésére. A kivételek fontos szerepet kapnak a virtuális módú taszkokat támogató védett módú programokban is.

3.2. Multitaszking

A multitaszking segítségével egyszerre több program (taszk) is futtatható egy processzoron. Legalábbis látszólag így tűnhet, a valóságban azonban az egyes taszkok csak a másodperc tört részéig kapnak vezérlést, és egy bizonyos idő letelte után vagy egy esemény bekövetkezésekor taszkváltás történik. Mivel a taszk által kapott időszeletek nagyon kicsik (század-, vagy ezredmásodpercek), ezért ez az időosztásos (time sharing) módszer azt a benyomást kelti, mintha több taszk futna párhuzamosan.

A multitaszking használata opcionális, és a legtöbb védett módú program esetén nincs szükség külön taszkok létrehozására.

3.2.1. A multitaszking operációs rendszerek típusai

A multitaszking operációs rendszereket többféle szempont szerint csoportosíthatjuk. Lehet egyrészt az alapján, hogy a taszkok vagy pedig az operációs rendszer határozza-e meg a vezérlésátadást.

A régebbi típusú operációs rendszerek esetén az egyes taszkok együttműködnek egymással, és ők határozzák meg, hogy mikor adják át a vezérlést (cooperativ multitasking). Ez persze veszélyes dolog, mivel ha egy program nem adja át a vezérlést egy másik taszk számára, akkor ez súlyos problémákat okozhat. Ezen az elven alapul például a Windows 3.1. A homokóra képe bizonyára mindenki számára ismerős. Miközben a homokszemek „peregnek", csak az adott taszk fut (illetve töltődik be), és lényegében nem történik taszkváltás.

Ennél korszerűbb módszer a pre-emptiv multitaszking, vagyis amikor az operációs rendszer dönti el, hogy mikor melyik taszk futhat, és mennyi ideig. Ezzel a megoldással sokkal nagyobb hatékonyság érhető el, ekkor a felhasználónak például már nem kell a homokórát bámulnia egy program betöltése közben. Ezen az elven alapul többek között a Linux, az OS/2, és a Windows NT. A Windows 95 a kompatibilitás miatt hibrid megoldást alkalmaz, mivel a 16 és 32 bites részek keverednek benne, így a pre-emptiv mellett megtalálhatóak benne a cooperativ multitaszking maradványai is, ami természetesen jelentős hatásfok-csökkenést eredményez.

Egy másik osztályozási szempont az, hogyan dönti el az OS, hogy melyik lesz a következő taszk, amelyik vezérlést kap.

A legelterjedtebb és a legkönnyebben megvalósítható módszer a „körben járás" (round-robin). Ekkor minden taszk egy meghatározott időszeletet kap a futásra, és a megszabott idő lejárta után az operációs rendszer a taszklista alapján automatikusan a következő taszkra kapcsol. Ez a lista egy kört alkot, az utolsó taszk után újra az első következik. Ha pedig egy új taszkot indítunk el, akkor az a lánc végére kerül.

Például, ha négy folyamat (A, B, C és D) fut egyszerre, és a megadott időszelet 10 ms, akkor a taszkok az ABCDABCD... sorrendben követik egymást, és mindegyik taszkra 25 alkalommal kerül sor 1 másodperc alatt.

Már ennél a módszernél is lehetőség van a hatásfok növelésére, többek között azzal, hogy egy billentyűre, vagy egyéb eseményre váró taszkot kizárunk ebből a körből. A fenti példánál maradva, ha a 'C' taszk egy file-t akar beolvasni, akkor mindaddig nem kap vezérlést, amíg a kívánt szektorok be nem töltődnek a háttértárról a memóriába. Ekkor a taszklista ABDABD... lesz. Bár ezt csak ritkán alkalmazzák, általában a többi taszk nem kap addig vezérlést, amíg a 'C' taszk be nem olvasta a file-t. Ennek megvalósításához az szükséges, hogy a lemezműveletek, és egyéb hardvereszközök kezelése teljesen a háttérben folyhasson, és függetlenek legyenek attól, hogy éppen melyik taszk fut.

A round-robin módszernél sokkal hatékonyabb a prioritáson alapuló multitaszking. Ekkor minden taszkhoz hozzárendelünk egy vagy több számot, amelyek a taszk fontosságára, a felhasznált erőforrásokra, stb. utalnak. A magasabb prioritású taszkok tehát nagyobb időszeletet kapnak a teljes időből. Az előtérben futó taszk prioritását az operációs rendszerek általában megnövelik, és amikor háttérbe kerül, akkor újra lecsökkentik. Ezáltal a felhasználóval jobban elhitethető, hogy több program fut egyszerre. A hatásfok tovább növelhető, ha például az aktuális taszknál nagyobb prioritású taszkot indítunk el, akkor az új azonnal megkapja a vezérlést, kiszorítva a régit. Az egyes taszkokat sorokba (queue) is lehet rendezni, az aktuális állapotuktól függően (például futtathatóak, egy időpontra vagy eseményre várakoznak, stb.).

A gyakorlatban a fenti két típust alkalmazzák a legtöbb operációs rendszerben, de ezeken kívül persze léteznek egyéb bevált módszerek is.

3.2.2. A hardver és a szoftver multitaszking

A processzor által támogatott multitaszking a következő rendszerobjektumok segítségével valósítható meg:

Taszk-állapotszegmens (TSS)	A processzor állapotának elmentésére és visszaállítására szolgáló szegmens
TSS deszkriptor	A TSS báziscímét és méretét tartalmazó deszkriptor, csak a GDT-ben lehet.
Taszk regiszter	Az aktuális taszk TSS-ének szelektorát tartalmazza
Taszkkapu deszkriptorok	A taszkváltások megkönnyítésére és szabályozására szolgál.

3.7. ábra A multitaszkinghoz szükséges rendszerobjektumok

A taszkváltások során a teljes környezet is megváltozik, vagyis a régi taszk állapota elmentődik, és az új állapota töltődik be. Ez a környezetváltás viszont elég sok időt vesz igénybe, ezért néhány létező operációs rendszer nem is használja a processzor által kínált szolgáltatást, hanem saját maga végzi el az információk elmentését és visszaállítását, illetve a „taszkváltást" (szoftver multitaszking). Például, ha egy egyszerűbb multitaszking operációs rendszer nem használja a lapozást, a lokális deszkriptortáblázatokat és a négy privilégiumszintet, az elmentendő információk mennyisége sokkal kevesebb a hardver taszkváltáshoz képest. Ráadásul néhány MOV utasítás az órajelek alapján jelentősen gyorsabb, mint egy teljes taszkváltás.

3.2.3. A taszk-állapotszegmens

Minden egyes taszkváltásnál szükség van a taszkok pillanatnyi állapotának elmentésére, illetve visszaállítására. Erre a célra szolgál a taszk-állapotszegmens (TSS), amely a processzor teljes állapotának megőrzésére szolgál, miközben egy másik folyamat kap vezérlést. A TSS segítségével tehát minden taszk számára a többitől független, önálló környezet alakítható ki.

A TSS mérete minimum 104 byte, az egyes mezők tartalma a 3.8. ábrán szerepel. A taszk-állapotszegmens ezen felül tartalmazhat I/O engedélyező bittérképet vagy egyéb adatokat is.

	0	15	31
0	Back Link		
4	ESP0		
8	SS0		
12	ESP1		
16	SS1		
20	ESP2		
24	SS2		
28	CR3		
32	EIP		
36	EFLAGS		
40	EAX		
44	ECX		
48	EDX		
52	EBX		
56	ESP		
60	EBP		
64	ESI		
68	EDI		
72	ES		
76	CS		
80	SS		
84	DS		
88	FS		
92	GS		
96	LDT szelektor		
100	Trap (0. bit)	I/O térkép címe	

3.8. ábra A taszk-állapotszegmens

Ha a taszkkapcsolást egy távoli hívás, egy megszakítás vagy egy kivétel okozta, akkor a processzor a régi taszk TSS-ének szelektorát elmenti az új taszk-állapotszegmensének Back Link mezőjébe, valamint az EFLAGS-ben lévő NT (Nested Task) bitet 1-re állítja, jelezve a taszkok egymásba ágyazását. A megszakításból való visszatérés során (IRET), ha az NT bit 1, akkor taszkváltás történik a Back Link-ben lévő taszkra; ha az NT = 0, akkor pedig egyszerűen csak a veremben lévő kódszegmensre adódik vissza a vezérlés (taszkváltás nélkül).

A FAR CALL utasítással hívott eljárásokból általában a RETF utasítással kell visszatérni. Ekkor viszont az NT bit nem kerül ellenőrzésre, tehát ha egy taszkot FAR CALL-al hívunk meg, akkor az IRET utasítással lehet visszakapcsolni a TSS legelején tárolt hívó taszkra.

A TSS tartalmazza mind a négy privilégiumszinthez tartozó verem címét is. Ha például egy 3. szinten futó taszk meghív egy 0. szinten lévő rutint, akkor az SS és ESP regiszterekbe a TSS-ben található SS0 és ESP0 értéke töltődik be, majd a 3. szintre való visszatéréskor az SS:ESP-be újra az alkalmazói program vermének címe kerül. Minden taszk számára annyi külön vermet kell biztosítani, ahány privilégiumszint használatos az adott rendszerben.

A TSS-ben található a CR3 regiszter tükörképe is, tehát ha a lapozás engedélyezve van, akkor ennek segítségével minden egyes taszkhoz külön lapcímtár rendelhető, és minden taszkváltás során a CR3-ba az adott taszk lapcímtára töltődik be. Például ennek segítségével könnyedén létrehozhatóak „virtuális gépek" is, elsősorban V86 módban. A V86-os taszkok csak 20 bites címet tudnak képezni, amely az alsó 1 MB alatti területre korlátozná az elhelyezkedésüket. A lapozás segítségével viszont a 20 bites lineáris cím a 4 GB-os területen belül bárhová leképezhető. Ezáltal minden virtuális módú taszk „úgy látja", mintha az 1 MB alatti területen helyezkedne el, de valójában bárhol lehet a 4 GB-on belül.

A CR3 regiszterhez hasonlóan minden taszk számára külön LDT állítható be (ezért lokális), vagy akár több taszk is használhatja ugyanazt az LDT-t. A taszkváltások során tehát az LDTR regiszterbe az TSS-ben lévő LDT szelektora töltődik be.

Igény szerint a TSS tartalmazhatja az I/O engedélyező bittérképet is, amivel az egyes I/O portokhoz való hozzáférés szabályozható. Minden bit megfelel egy portnak, és mivel 65536 port van, ezért a bittérkép maximális mérete 8192 byte. A bittérkép TSS-en belüli kezdetét az I/O térkép címe, a végét pedig a taszk-állapotszegmens mérete határozza meg. Ha egy virtuális módú vagy az IOPL értékénél nagyobb privilégiumszintű taszk (CPL > IOPL) egy I/O műveletet hajt végre, akkor az I/O engedélyező térképben az adott porthoz tartozó bit ellenőrzésre kerül. Ha ez a bit 0, akkor az I/O művelet végrehajtható, ha 1, akkor általános védelmi hiba keletkezik. Ugyancsak kivétel gene-

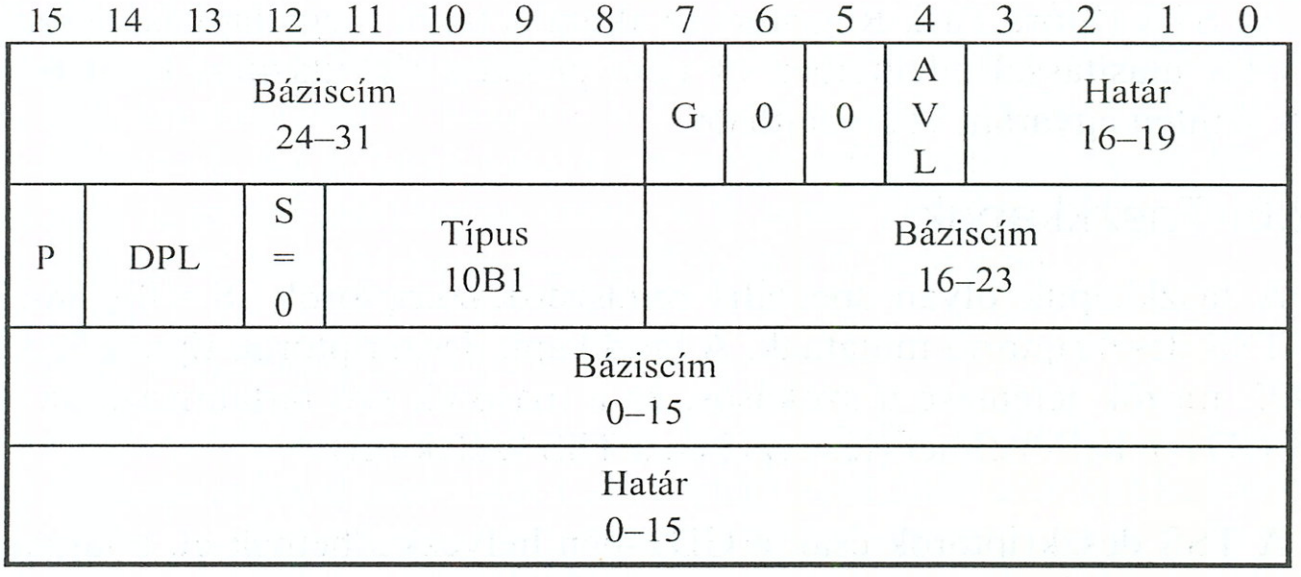
rálódik, ha az adott bit a térkép határán kívül esik. Az összes port elérése letiltható, ha a bittérkép báziscíme a TSS méretén túlra mutat.

Tegyük fel, hogy egy operációs rendszeren belül az IOPL = 1, így a 0. privilégiumszinten futó kernel, és az 1. szinten lévő driverek elérik a szükséges I/O portokat. Ha viszont egy V86-os, illetve egy 2. vagy 3. szintű taszk próbálkozik például a közkedvelt in al, 61h utasítással, és az I/O engedélyező bittérkép mérete kisebb, mint 12, vagy a 12. byte 1. bitje 1-re van állítva, akkor általános védelmi hiba keletkezik. A kivételkezelő segítségével az operációs rendszernek lehetősége van arra is, hogy az I/O műveletet emulálja, és visszaadja a vezérlést a taszknak, vagy esetleg megszakítsa a taszk futását.

Ha taszkváltás során az új taszk TSS-ében a 100. byte legalsó bitje (Trap bit) 1-re van állítva, akkor debug kivétel (INT 1) keletkezik, és ezáltal a taszkváltásokat is nyomon lehet követni.

3.2.4. A TSS deszkriptor

A többi szegmenshez hasonlóan a taszk-állapotszegmenshez is tartozik egy deszkriptor, a TSS deszkriptor. Ez rendszerdeszkriptor, és csak a GDT-ben helyezkedhet el. A TSS deszkriptor tartalmazza a TSS 32 bites báziscímét és 20 bites méretét, valamint a P bitet (jelen van-e), és a DPL mezőt (mely szintekről érhető el).



3.9. ábra A TSS deszkriptorok felépítése

A taszkok foglaltságát a TSS deszkriptor B (Busy) bitje jelzi, amely megakadályozza, hogy olyan taszkra adjuk át a vezérlést, amely már korábban meghívott egy másik taszkot, vagy amely meg lett szakítva. Egy szabad TSS deszkriptor típusa 9, egy foglalté pedig 11.

A taszkok tehát nem ágyazhatóak korlátlanul egymásba. Ha például az 'A' taszk a FAR CALL utasítással meghívja a 'B' taszkot, a 'B' pedig a 'C'-t, akkor a 'C' taszk nem hívhatja meg sem 'A'-t, sem 'B'-t (sem önmagát), különben kivétel keletkezik.

3.2.5. A taszkregiszter

A taszkregiszter (TR) az aktuális taszk TSS szelektorát tartalmazza. A TR tartalmának megvizsgálására és megváltoztatására az LTR és az STR utasítások szolgálnak.

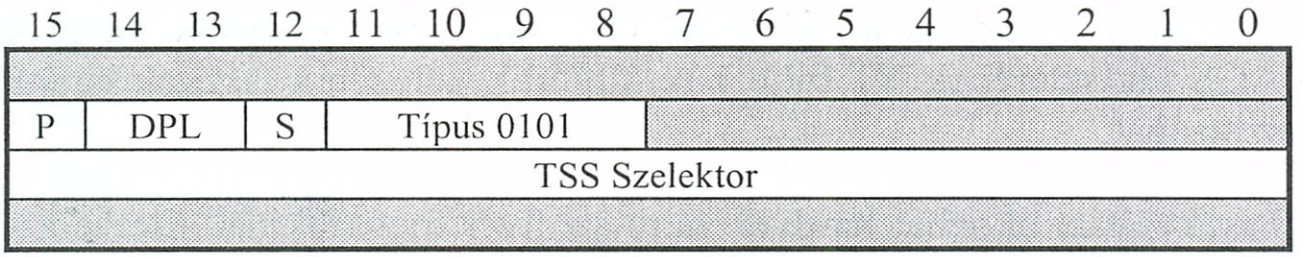
Az LTR (Load Task Register) utasítást csak a 0. privilégiumszinten lehet végrehajtani, és elsősorban a védett módba kapcsolás után, de még az első taszkváltás előtt szükséges, azért, hogy kiválasszuk azt a TSS-t, ahová az első taszváltás során a processzor állapota elmentődik. Az LTR-re akkor is szükség van, ha több privilégiumszintet használunk, de csak egy taszk van, és taszkváltás egyáltalán nem is történik. Ugyanis amikor magasabb privilégiumszintű rutint hívunk meg, akkor a veremváltás során az SS:ESP-be a TSS-ben lévő adott szinthez tartozó verem szelektora és ofszetcíme töltődik be.

Az STR (Store Task Register) az aktuális taszk megállapítására szolgál. Az LTR utasítással ellentétben ez nem privilegizált utasítás, tehát mind a négy szinten egyaránt végrehajtható.

3.2.6. Taszkkapuk

A taszkkapuk olyan speciális rendszerdeszkriptorok (S = 0), amelyek egy TSS deszkriptorra mutatnak. A taszkkapu deszkriptorok típusa 5, a P és a DPL mezők jelentése a szokásos, és a második szó tartalmazza az adott TSS GDT-n belüli címét (lényegében a TSS szelektort).

A TSS deszkriptorok csak a GDT-ben helyezkedhetnek el, a taszkkapu deszkriptorok viszont az IDT-ben és az LDT-ben is.



3.10. ábra A taszkkapu deszkriptorok felépítése

Ezáltal egyrészt lehetőség van arra, hogy egy megszakítás vagy egy kivétel is taszkváltást okozzon. Például a veremhiba (12. kivétel) kezelését igazán eredményesen csak egy külön taszkként lehet megvalósítani. Ugyanis ha a 0. privilégiumszinten betelik a verem, a kivételkezelő sem tud meghívódni, hisz az INT 12 meghívásához a veremben további 3 duplaszóra (EFLAGS, CS, EIP) lenne szükség. Ez tehát dupla, majd tripla hibához vezet. Viszont ha egy taszkkapcsolás történik, akkor már az SS:ESP egy érvényes veremre mutat, és így elhárítható a hiba a (lefelé bővülő) betelt verem megnövelésével vagy a hibát okozó taszk futásának megszakításával.

Másrészt a taszkkapukkal az is szabályozható, hogy az egyes taszkok milyen más taszkokat hívhatnak meg. Először vizsgáljuk meg, mi történik, ha adott három alkalmazói szintű taszk (A, B, C), és a multitaszking megvalósításában csak a TSS-eket használjuk. Ahhoz, hogy az 'A' taszk meghívhassa a 'B'-t, a 'B' taszk TSS deszkriptorában a DPL mezőnek háromnak kell lennie. Mivel a 'C' taszk is a 3. szinten van, ezért ő is elérheti a 'B' taszk TSS-t, vagyis illegálisan átkapcsolhat a 'B'-re.

Ezt csak a taszkkapuk segítségével lehet megakadályozni. Ehhez a taszkok TSS deszkriptorát 0. szintűre kell állítani, hogy egyik 3. szintű taszk sem érhesse el. Az A taszk lokális deszkriptortáblázatában pedig létre kell hozni egy 3. szintű taszkkaput, amely a 'B' taszk TSS-ére mutat. Vagyis ezáltal csak az 'A' taszk hívhatja meg a 'B'-t, de a 'C' már nem.

3.2.7. A taszkkapcsolás

A taszkváltás leggyakoribb módja, ha az aktuális taszk egy TSS-re vagy egy taszkkapura hivatkozó FAR CALL vagy FAR JMP utasítást hajt végre.

Ha a taszkváltást a FAR JMP idézte elő, akkor a régi TSS deszkriptor Busy bitje törlődik, és az új taszk lesz foglaltnak jelölve (B = 1). A FAR

CALL esetén a régi és az új taszk is foglalt lesz, valamint az új taszk EFLAGS regiszterének NT bitje 1-re állítódik, jelezve a taszkok egymásba ágyazottságát.

Taszkváltást okozhat hardver vagy szoftver megszakítás, illetve kivétel is, ha az IDT-ben az adott megszakításhoz egy taszkkapu deszkriptor (típus = 5) tartozik. A FAR CALL-hoz hasonlóan a hívó és a hívott taszk TSS deszkriptorában a B bit 1 lesz, és az új taszk NT bitje szintén 1-re állítódik.

Egy meghívott taszkból az IRET utasítással lehet visszatérni a hívóhoz. Ha az EFLAGS-ban az NT bit 1, akkor nem a veremben lévő CS:EIP-re adódik át a vezérlés, hanem egy taszkváltás történik a taszk-állapotszegmens Back Link mezőjében lévő TSS szelektorra, és egyúttal a régi taszk foglaltsága megszűnik (B = 0)

A megszakításokból tehát az IRET utasítással lehet visszatérni, függetlenül attól, hogy a megszakítás-kezelő külön taszkként vagy csak egyszerű rutinként lett megvalósítva. A FAR CALL használatakor azonban már van eltérés, ugyanis ha egy másik szegmensben lévő távoli eljárást hívunk meg, akkor a RETF utasítással, ha viszont egy másik taszkot hívunk meg, akkor az IRET utasítással kell visszatérni (a RETF nem ellenőrzi az NT bitet, és taszkváltásra sem alkalmazható).

A hívás módja	A visszatérés módja
egyszerű megszakítás	IRET
megszakítás taszkkapun keresztül	IRET
szegmensek közötti FAR CALL	RETF
taszkok közötti FAR CALL	IRET

Szükség lehet arra, hogy ugyanazt a távoli eljárást szegmensek közötti és taszkok közötti FAR CALL-lal is meghívjuk. A hívott eljárás pedig nem tudhatja, hogy milyen módszerrel hívták meg, illetve hogy a RETF vagy az IRET utasítással térjen-e vissza. Ez a probléma úgy oldható meg, hogy minden szegmensek közötti hívás előtt a EFLAGS-ot a verembe helyezzük, és ekkor már a taszkok közötti híváshoz hasonlóan az IRET használható a visszatérésre.

```
; Az eljárás meghívása külön taszkként

A SEGMENT

...
CALL FAR C_TSS_SEL:C_PROC
...

A ENDS

; Az eljárás meghívása szegmensek közötti hívással
B SEGMENT

...
PUSH EFLAGS
CALL FAR C_CODE_SEL:C_PROC
...

B ENDS

; az NT bittõl függõen visszatérés az A-ra vagy a B-re
C SEGMENT

C_PROC:
    ...
    IRETD

C ENDS
```

Minden taszkváltás során a CR0 regiszter TS (Task Switched) bitje 1-re állítódik. Ez a bit törölhető CLTS utasítással, vagy közvetlenül a CR0 regiszterben. Ha egy taszk egy koprocesszor műveletet hajt végre, miközben a TS = 1, akkor a 7. kivétel (koprocesszor nem elérhető) generálódik. Ezzel a módszerrel nem kell minden egyes taszkváltáskor a koprocesszor teljes állapotát elmenteni, hanem csak akkor, amikor szükség van rá. A koprocesszor állapota akár tárolható a TSS-ben is, a 104. byte és az esetleges I/O engedélyező bittérkép közötti területen. Az ugyancsak 104 byte-os állapot elmentésére az FSAVE, visszaállítására az FRSTOR utasítások használhatóak.

3.3. Lapozás

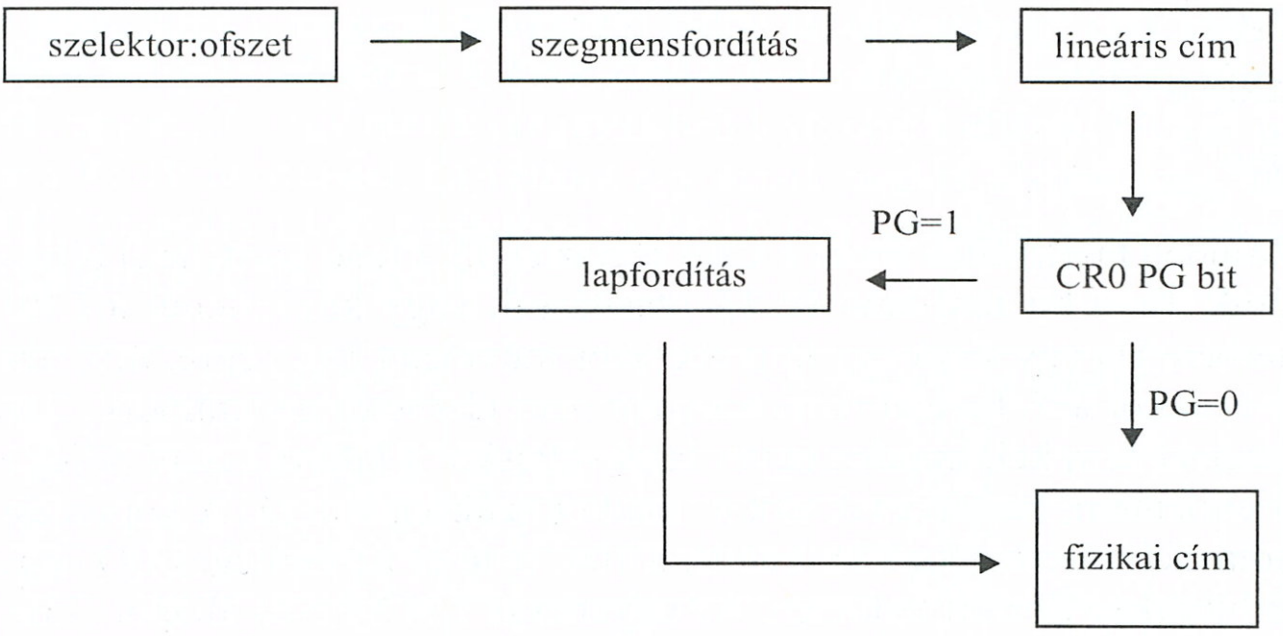
A lapozás (paging) elve egyszerű: a teljes 4 GB-os memóriatartományt szétszabdaljuk 4 KB-os lapokra, majd ezeket a lapokat igény szerint újra összerakjuk. Ennek segítségével megvalósítható többek között a virtuális

memória vagy egyszerre több virtuális módú taszk futtattása az 1 MB alatti területen.

A lapozást a CR0 regiszter PG bitjének 1-re állításával engedélyezhetjük, általában a védett módba való átkapcsolás és egyéb inicializálás után. A lapozás használata a multitaszkinghoz hasonlóan opcionális, vagyis az egyszerűbb védett módú programokban nincs rá szükség, viszont az engedélyezésével a processzorban rejlő lehetőségek jobban kiaknázhatóak.

3.3.1. Lapfordítás

Védett módban a szelektor:ofszet formájú címekből a szegmensfordítás során (deszkriptor bázis + ofszetcím) egy 32 bites lineáris cím keletkezik. Ha a PG bit 0, akkor ez a lineáris cím megegyezik fizikai címmel. Ha a lapozás engedélyezve van, akkor a lapozó egység a lineáris címet laptáblázatok segítségével képezi le fizikai címmé.



3.11. ábra Fizikai cím képzése

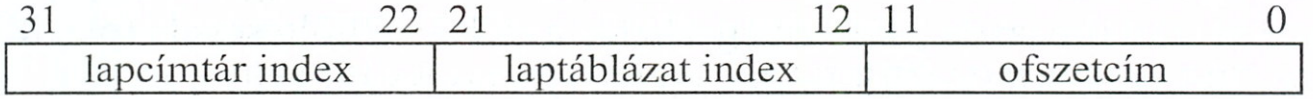
A lapozás kétféle adatstruktúrára épül: a lapcímtárakra (page directory), és a laptáblázatokra (page table). A lapcímtárak bejegyzései a laptáblázatokra mutatnak, a laptáblázatok bejegyzései pedig az egyes lapokra.

Mindkét táblázattípus 4 KB méretű, és mindkettőben 1024 darab 32 bites bejegyzés található. Tehát a lapcímtárak 1024 laptáblázatra mutatnak, és

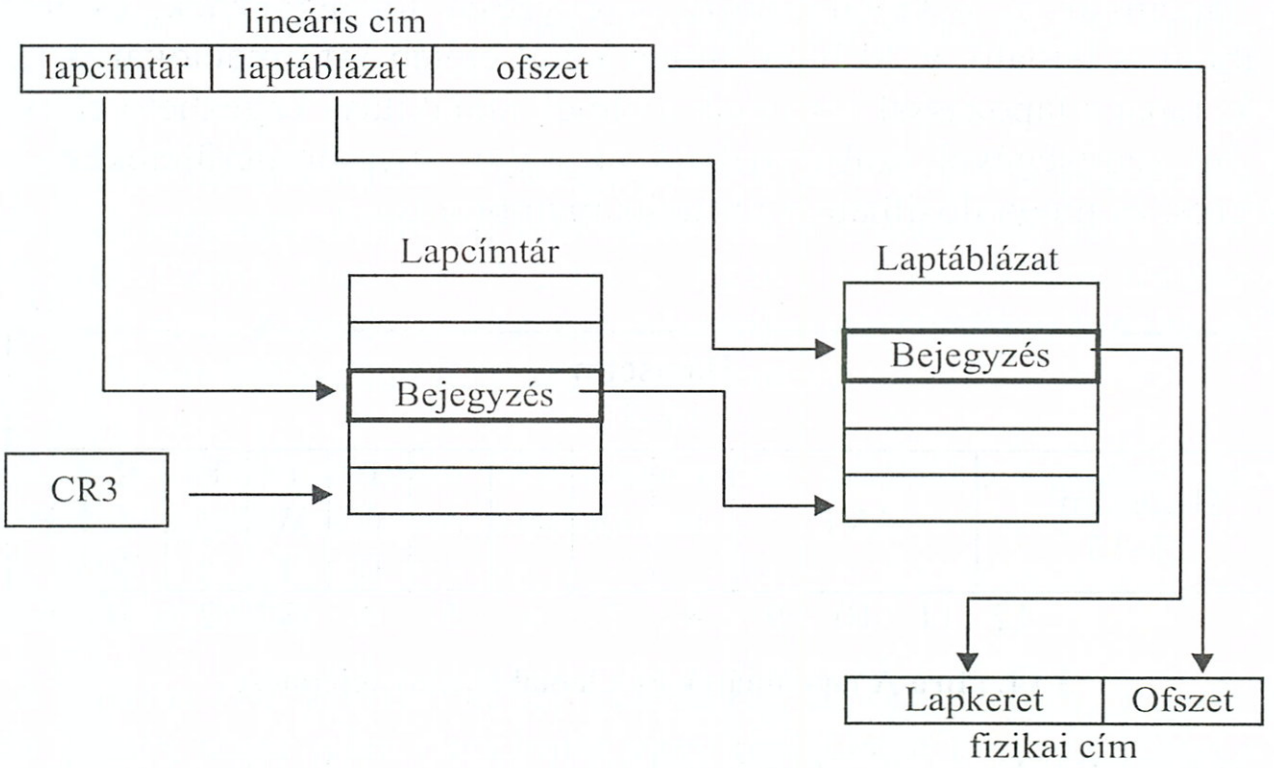
minden egyes laptáblázat 1024 darab 4 KB-os lapkeret címét tartalmazza, így a kétszintű táblázatokkal a teljes 4 GB lefedhető (1024*1024*4096).

Az aktuális lapcímtár fizikai báziscímét a CR3 regiszter tartalmazza. Mivel a taszk-állapotszegmensekben a CR3 is megtalálható, ezért akár minden taszk számára külön lapcímtár alakítható ki.

Lapozás esetén a 32 bites lineáris cím a 3.12. ábrán látható módon három részből tevődik össze: egy 10 bites lapcímtár indexből, egy 10 bites laptáblázat indexből és egy 12 bites ofszetből.



3.12. ábra A lineáris cím felépítése



3.13. ábra A lineáris cím átalakítása fizikai címmé

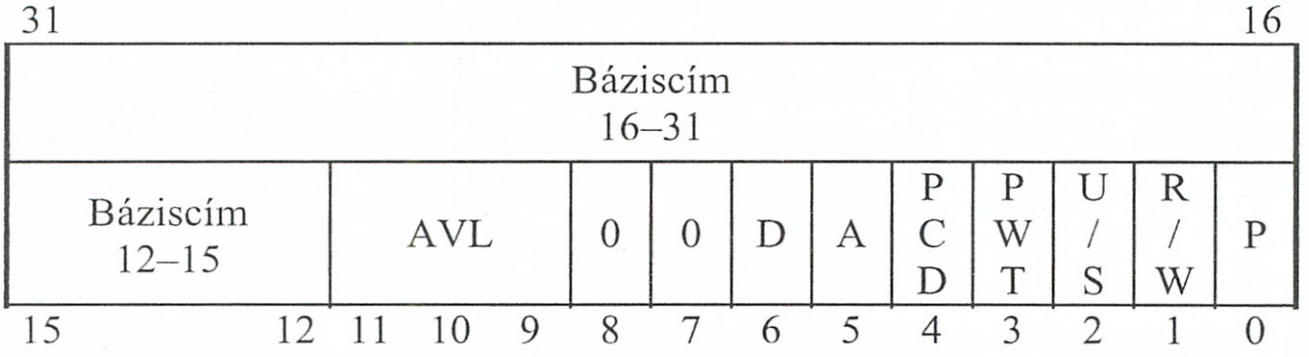
A lapcímtár és a laptáblázat indexek egy táblázaton belüli elemre mutatnak, ugyanúgy, ahogy például a szelektorok egy deszkriptortáblázaton belüli elemeket választanak ki. A lapcímtár index az aktuális (CR3-ban lévő báziscímű) lapcímtár egy bejegyzésére hivatkozik, amely tartalmazza a megfelelő

laptáblázat címének felső 20 bitjét. (Minden laptáblázatnak és lapnak 4 KB-os határon kell elhelyezkednie, ezért az alsó 12 bit felesleges.) A laptáblázat index pedig a megadott laptáblázat egy bejegyzésére mutat, amelyben a lap fizikai címének a legfelső 20 bitje szerepel. Végül ennek a 20 bitnek és a lineáris címben lévő 12 bites ofszetcímnek az összegéből keletkezik a 32 bites fizikai cím.

Mivel a címátalakítás viszonylag lassú folyamat, ezért a processzorban található egy lapfordítást segítő cache, a TLB (Translation Lookaside Buffer), amely a leggyakrabban használt címeket tartalmazza. A laptáblázatok változtatásakor szükség van a TLB törlésére is, hogy mindig pontos adatokat tartalmazzon. Ez elvégezhető a CR3 feltöltésével (pl. MOV EAX,CR3 majd MOV CR3,EAX), illetve egy taszkváltással.

3.3.2. Laptáblázat bejegyzések

A lapcímtár és a laptáblázatok bejegyzései megegyeznek, és mint a 3.14. ábrán is látható, a 20 bites címen kívül egyéb információkat is tartalmaznak, tehát a lapozással nem csak a címek átalakítása végezhető el. Az alsó tíz bit segítségével szabályozható az egyes memóriaterületekhez való hozzáférés, és megvalósítható a virtuális memória is.



3.14. ábra A lapcímtárak és a laptáblázatok felépítése

A P bit jelzi, hogy az adott laptáblázat vagy az adott lap jelen van-e a memóriában. A P bit törlésével egyrészt megadhatóak azok a lapok, amelyek a háttértáron vannak, másrészt amelyeket az adott taszk nem érhet el. A lapcímtárban lévő P bit 0-ra állításával teljes 4 MB-os területeket is elérhetetlennek lehet jelölni, és ekkor a laptáblázatnak sem kell a memóriában lennie.

Ha egy olyan bejegyzésre történik hivatkozás, amelyben a P bit értéke 0, akkor laphiba keletkezik. A kivételkezelő ekkor betöltheti a háttértárról a kívánt lapot (egy másik lap felülírásával), majd visszaadhatja a vezérlést a hibát okozó utasításra.

Az R/W bit (Read/Write) funkciója az írásvédelem. Ha 0, akkor a lap csak olvasható, ha 1, akkor írható és olvasható is. A U/S bitnek (User/Supervisor) ugyancsak a lapszintű védelemben van szerepe. Ha ez a bit magasra van állítva, akkor az adott lap minden privilégiumszintről elérhető, ha törölve van, akkor a felhasználói szintről nem érhető el, csak a 0., az 1. és a 2. szintekről. Ha írás történik egy csak olvasható lapra, vagy ha egy alkalmazói program egy U/S = 0 lapot próbál meg elérni, akkor a 14. kivétel jön létre. (Bár ha a CR0 regiszter WP bitje 0, akkor felügyelői szinten a csak olvasható lapokba is lehet írni).

A PWT (Page Write-Through) és a PCD (Page Cache Dissable) bitek a 80486-osnál jelentek meg először, és a lapszintű cache működését befolyásolják.

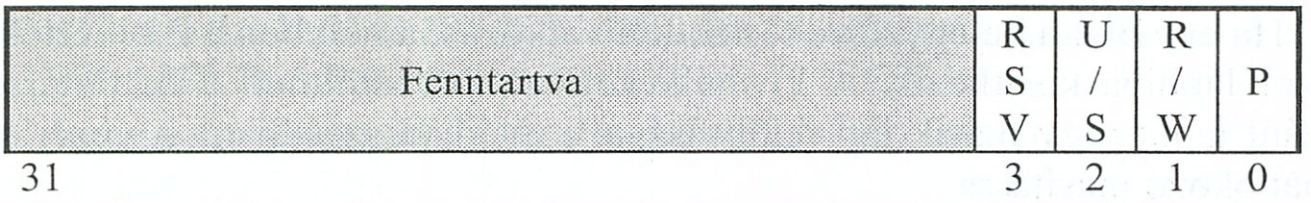
A lapok használatakor a bejegyzések 5. és 6. bitjét a processzor automatikusan magasra állítja: olvasás esetén az A (Accessed) bitet, írás esetén pedig a D (Dirty), illetve az A bitet is. Ezen bitek periodikus törlésével kiszűrhetőek azok a lapok, amelyeket célszerű a memóriában tartani, és azok, amelyeket hely hiányában lemezre lehet menteni.

A 9–11. bitek szabadon felhasználhatóak, például a lapozás teljesítményét növelő egyéb információk tárolására.

3.3.3. A laphiba

A laphiba (14. kivétel) akkor keletkezik, ha olyan lapra történik hivatkozás, amely nincs jelen (lapcímtár vagy laptáblázat bejegyzésében a P bit 0), vagy ha lapszintű védelemsértés történt (írásvédett lapra írás, vagy egy felügyelői lap elérése felhasználói szinten).

A hiba oka pontosan azonosítható, mivel a hibát okozó lineáris cím a CR2 regiszterbe kerül, és a verembe kerülő hibakód is sok információt tartalmaz.



3.15. ábra A laphiba kivétel hibakódja

A hibakódban a 0–2. bitek hasonlítanak a laptáblázatok bejegyzéseire. Ha a P bit 0, akkor a hibát egy nem jelenlévő lap (vagy laptáblázat) okozta, ha a bit 1, akkor védelemsértés történt. Az R/W bit jelzi, hogy olvasás (0) vagy írás (1) miatt keletkezett a hiba. Ha a U/S bit 1, akkor a 3. szinten történt a hiba, ha 0, akkor pedig a felügyelői szinten. A Pentium processzortól kezdve a 3. bit is definiált, ennek értéke akkor 1, ha valamelyik bejegyzésben egy fenntartott bit lett megváltoztatva.

3.3.4. A lapozás jellemzői

Egy fejlett lapozást használó rendszer kialakítását kicsit megnehezíti, hogy a lapozás két szintre épül: a lapcímtárra és a laptáblázatra. Viszont ez a módszer sokkal gazdaságosabb, mivel így a lapcímtárakban a P bitek törlésével egyszerre 4 MB-os területek jelölhetők elérhetetlennek, és ekkor persze a 2. szinten lévő laptáblázatokat sem kell definiálni, vagy akár lemezre is menthetők.

Ha például a lapozás csak egyszintű lenne, akkor egy maximálisan 4 GB-os virtuális memória kialakításához 4 MB fizikai memóriára lenne szükség. Ha egy 16 MB-os gépben szeretnénk a fizikai memóriát leképezni, akkor az egyszintű lapozás esetén a memória 1/4 részét a laptáblázatok mindenképpen felemésztenék, a kétszintű lapozással pedig mindössze egy 4 KB-os lapcímtárra és 16 KB-os laptáblázatokra (4*4 KB) van szükség. Mindezek csak egyetlen taszkra vonatkoznak, így könnyen belátható, hogy az egyszintű laptáblázatok alkalmazása célszerűtlen lenne.

Mivel a 4 GB-os címek szabadon leképezhetőek a fizikai memóriára, ezért érdekes címek alakíthatóak ki, például egy program futtatható a 2 GB-os területen, vagy a grafikus képernyő áthelyezhető az 512 MB-os címre.

Az is pozitívum, hogy a lapozás teljesen áttetsző a felhasználói programok számára, hisz számukra nincs különbség, hogy egy 32 bites ofszetcímet

használnak, vagy egy 3 részből álló virtuális címet. Ha pedig egy olyan lapra hivatkoznak, amely a háttértáron van, akkor a kivételkezelő betölti az adott lapot, és erről az alkalmazói program szintén nem szerez tudomást.

Továbbá a lapozás segítségével a szegmentálás számára egybefüggő terület szimulálható, mivel a lapfordítás a szegmensfordítás után következik. Ha például egy program lefoglal egy 5 MB-os területet, akkor azt egy egybefüggő szegmensként látja, de a fizikai memóriában ez akár 50 kisebb szétszórt területből is állhat, ami csak a lapozás által tűnik egységes területnek, valamint a szegmens egy részének akár a memóriában sem kell lennie.

A lapozás engedélyezésével a védelmi rendszer hatékonysága is fokozható a lapszintű és a szegmensszintű védelem kombinálásával. Egyrészt a P bittel nem csak azokat a lapokat lehet jelezni, amelyek el lettek távolítva a fizikai memóriából, hanem azokat is, amelyekhez az adott taszknak nincsen hozzáférési joga. Ha egy taszk esetleg hozzájutna egy olyan szelektorhoz, amellyel egy másik taszk területén is garázdálkodhat, akkor ez megakadályozható, ha minden taszk laptáblázataiban csak azok a lapok bejegyzései szerepelnek, amelyek ténylegesen az adott taszkhoz tartoznak. Ekkor egy másik taszk szegmensének használatakor laphiba keletkezik, és a kivételkezelő megszakíthatja a hibát okozó taszk futását.

A Read/Write bittel egyes lapok írásvédetté is tehetők, a User/Supervisor bittel pedig beállítható, hogy bizonyos lapokat az alkalmazói programok ne érjenek el, viszont a biztonságosabb szintek (a 0., az 1., és a 2. szint) már igen. Ezek főleg akkor hasznosak, ha az „ügyesebb" felhasználói programoknak sikerül valamilyen módon a szegmensszintű védelmet megkerülni, és olyan memóriaterülethez hozzáférni, amely a felsőbb szintekhez tartozik.

Akár arra is lehetőség van, hogy a védelmi mechanizmus csak a lapozáson keresztül fejtse ki hatását. Ez úgy érhető el, ha a taszkoknak olyan szegmenseket állítunk be, amelyek báziscíme 0, és a határ a maximális 4 GB. Minden taszk laptáblázatában a felhasználói program kód-, adat-, és veremszegmensek lapjaiban az U/S bitet 1-re kell állítani, a kernel (0. szint), és az esetleges driverek (1. szint) lapjaiban pedig 0-ra, illetve a többi lapot nem jelenlévőnek (P = 0) kell jelölni. Ekkor nincs szükség a szegmensregiszterek gyakori változtatására sem (ami védett módban elég lassú), és a

memóriafoglalásnál (vagy a verem növelésekor) is elég csak a laptáblázatokat módosítani, és nem kell új szegmenseket létrehozni vagy átméretezni. Tehát már ezzel az egyszerű módszerrel is elég hatásos rendszert lehet kialakítani.

3.4. A virtuális 8086-os üzemmód

A virtuális 8086-os mód használatával egy vagy több 8086-os program futtatható a védett módú környezet felügyelete alatt. Ilyen például a Windows alatti DOS-ablak (shell), vagy az EMS driver alatt futó programok.

Virtuális üzemmódban a memóriacímzés (szegmentálás) hasonlóan történik, mint a 8086-os processzoron, vagyis a 20 bites lineáris cím a szegmensregiszterekben lévő báziscím 16-szorosának és a 16 bites ofszetcímnek az összegéből adódik. A különbség csak annyi, hogy az alkalmazások a védett módú környezet alatt, a 3. privilégiumszinten futnak. Ezért eltérően a valós módtól, a virtuális módú programoknak a 3. privilégiumszint korlátai között (IOPL és privilégizált utasítások) kell futni.

A virtuális 8086-os üzemmódban – az elnevezés ellenére – a használható utasítások és regiszterek nem korlátozódnak a 8086-os processzor szintjére. Bár a virtuális mód működése (memóriacímzés, stb.) nagyon hasonlít a valós módhoz, de a 8086-os processzorral ellentétben a virtuális módban például a 32 bites általános regiszterek is használhatóak az utasításokban.

Ha egy kivétel vagy megszakítás keletkezik, akkor a processzor visszakapcsol védett módba. A kivételek és a megszakítás-kezelők segítségével a védett módú operációs rendszernek a következőkre nyílik lehetősége:

- az egyes regiszterek, a hardver, és az operációs rendszer adatainak védelme az illegális hozzáférésektől
- egy 16 bites operációs rendszer funkcióinak könnyebb emulálása a védett módú OS által.
- az osztott erőforrások könnyebb irányítása több V86-os és egyéb taszk között

3.4.1. A virtuális üzemmódú taszkok elindítása

Az EFLAGS regiszter VM bitje határozza meg, hogy egy taszk védett (VM = 0) vagy virtuális 8086 (VM = 1) üzemmódban módban fut. Egy virtuális módú taszk kétféleképpen indítható el: taszkváltással vagy az IRETD utasítással.

Egy virtuális üzemmódú taszk elindításának legegyszerűbb módja, ha az adott taszk TSS-ében az EFLAGS képében a VM bitet 1-re állítjuk. A TSS-ben lévő szegmensregiszterekben nem szelektorokat kell elhelyezni, hanem valós módú báziscímet (a 8086-os címzésmódnak megfelelően).

Mivel a VM bit csak a 0. privilégiumszinten változtatható meg az IRETD utasítással, ezért a második módszerrel csak a felügyelői szinten indítható el egy V86-os taszk. Ez a megoldás a virtuális módban keletkezett megszakításokból való visszatérés szimulálásán alapul. Ekkor a 3.. ábrán is látható módon a veremben el kell helyezni a CS:EIP regisztereket (belépési pont), az EFLAGS képét, amelyben a VM = 1, majd a taszk 3. szintű vermének SS:ESP mutatóját, végül a többi szegmensregisztert (GS, FS, DS, ES), amelyeknek valós módú báziscímeket kell tartalmazniuk.

3.4.2. A virtuális módú környezet emulálása

A virtuális módban futó programok működését alapvetően befolyásolja az IOPL értéke. Ha az IOPL kisebb, mint 3 (a virtuális módú alkalmazások a 3. privilégiumszinten futnak), akkor bizonyos utasítások végrehajtása általános védelmi hibát okoz. Ezek az érzékeny utasítások a CLI, STI, PUSHF, POPF, INT n, és az IRET. Az I/O utasítások (IN, OUT, INS, OUTS) viszont nem függenek az IOPL szinttől, ezek végrehajtásakor a processzor mindig ellenőrzi az I/O bittérképet, és ha az adott bit(ek) értéke 1, akkor kivételt generál, ha 0, akkor végrehajtja az I/O műveletet.

Miután a kivételkezelő megállapította, hogy egy V86 taszk okozta a hibát (a veremben lévő EFLAGS-ban a VM bit 1), lehetősége van, hogy emulálja ezeket az érzékeny utasításokat. A hibás utasítás címe ugyancsak a veremben van, viszont ez egy 8086 típusú (szegmens:ofszet) cím, ezért az adott utasítást a megfelelő címátalakítás után egy védett módú szelektoron keresztül kell elérni.

3.4.3. Az IF flag emulálása

Ha egy virtuális módú program letilthatná a megszakításokat, akkor az könnyen a teljes operációs rendszer összeomlásához vezethetne. Emiatt szükség van az összes olyan utasítás emulálására, amelyek megváltoztathatják az IF bitet.

Bár a PUSHF nincs hatással az IF bitre, de ennek emulálása is ajánlott, mivel enélkül a virtuális módú program könnyen rájöhet, hogy a megszakításokat nem tudja befolyásolni.

```
pushf                   ; a flagek a verembe
cli  ; megszakítások letiltása
pushf                   ; a flagek újra a verembe
pop  ax                 ; a két állapot az AX-be
pop  bx                 ; és a BX-be
cmp  ax,bx              ; megváltozott az IF bit?
jnz  IF_Emu             ; nem.
```

A CLI és az STI utasítások nyomonkövetésével tárolható a virtuális módú taszkok látszólagos IF flagje. Ha a kivételkezelő észleli, hogy a PUSHF okozta a hibát, akkor elhelyezhet a V86 taszk vermében egy olyan FLAGS másolatot, amelyben az IF bit állapota megfelel ennek a látszólagos értéknek. A POPF utasítás is emulálható, ha a kivételkezelő úgy módosítja a vermet, hogy a megszakításból való visszatéréskor a taszk FLAGS regisztere a megfelelő legyen. A PUSHF és a POPF esetén a 3. szintű veremmutatót (SP) is a megfelelő módon meg kell változtatni, mintha tényleg megtörtént volna a FLAGS regiszter elmentése vagy visszaállítása.

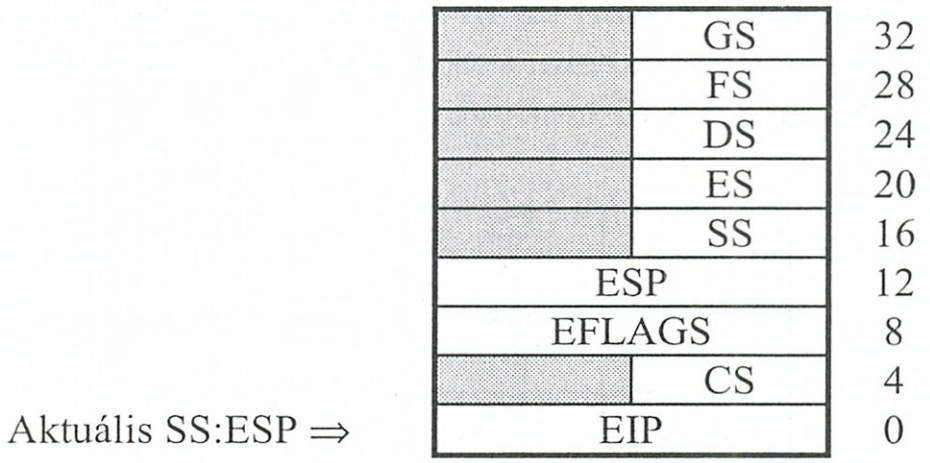
Mivel ennek az egyetlen bitnek a tökéletes emulálása elég körülményes feladat, és az időveszteség is jelentős, ezért a Pentium processzoroknál bevezették a virtuális megszakításokat. Ennek alkalmazásával már nincs szükség a felügyelőprogram állandó meghívására, mivel a V86 taszkok egy látszólagos megszakítás flaget (VIF) változtatnak meg az EFLAGS regiszterben, nem pedig az igazi IF bitet.

Ha a virtuális módú taszkok IOPL értéke 3, akkor az IF flag nyugodtan megváltoztatható, és a CLI, STI, PUSHF, POPF utasítások nem okoznak kivételt. Viszont ez könnyen problémákhoz vezethet, ezért a biztonság érdekében ajánlatosabb az utasítások emulálása.

3.4.4. A megszakítások működése és emulálása

Mivel védett módban a megszakítástáblázat felépítése a valós módhoz képest teljesen eltérő, ezért virtuális üzemmódban a megszakítások emulálására is szükség van.

Ha virtuális üzemmódban egy védett módú kivétel vagy megszakítás keletkezik, akkor a processzor átkapcsol védett módba, és a 0. szintű veremben szokásos módon elhelyezi az EFLAGS és a CS:EIP regisztereket, illetve néhány kivétel esetén a 32 bites hibakódot is. Mivel a privilégiumszint is megváltozik, ezért a 3. szintű SS:ESP regiszterek is a verembe kerülnek, és a TSS-ben lévő 0. szinthez tartozó értékek töltődnek be az SS:ESP-be. A szegmensregiszterekben lévő virtuális módú báziscímek védett módban használhatatlanok, ezért ezek is elmentődnek a veremben, a szegmensregiszterek pedig törlődnek (a GDT-ben lévő 0. deszkriptorra mutatnak). A szegmensregiszterek megfelelő szelektorokkal való feltöltése a kivételkezelő feladata. Egy virtuális módú megszakítás esetén a verem képe tehát az alábbi ábra szerint alakul (hibakód nélkül).



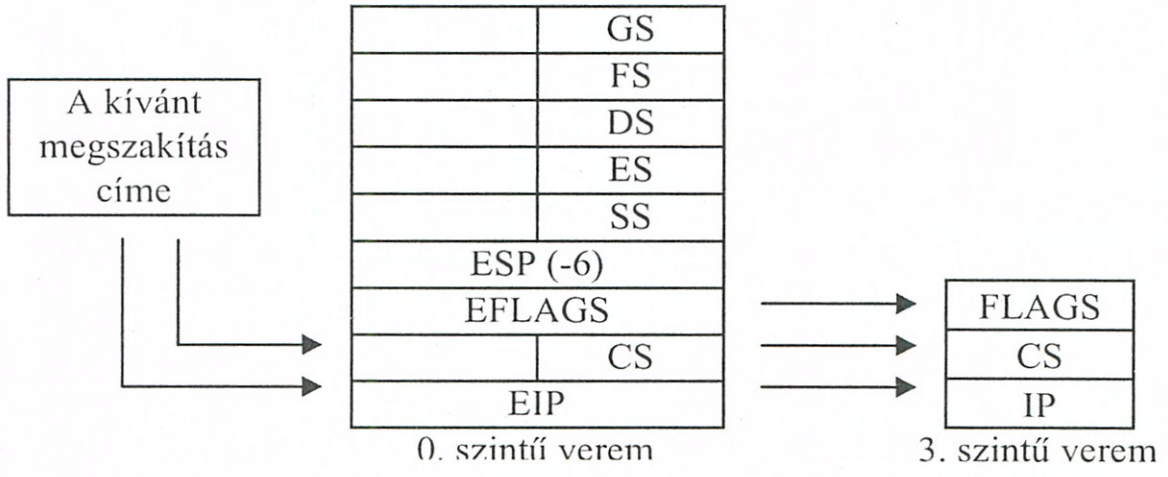
3.16. ábra A verembe mentett regiszterek V86-os megszakítás esetén

Ha az IRET utasítás a veremben olyan EFLAGS-ot talál, amelyben a VM bit 1, akkor feltételezi, hogy egy virtuális módú taszkra kell visszakapcsolni. Ekkor visszaállítja a 3. szintű verem címét az SS:ESP-ben, és betölti a szegmensregiszterekbe a régi báziscímeket, majd átadja a vezérlést a veremben lévő CS:EIP-re.

Az IF bithez hasonlóan a virtuális módú megszakításokat is az IOPL szint befolyásolja. Ha az IOPL = 3, akkor a virtuális módban végrehajtott INT n utasítások hatására a védett módú megszakítás-kezelők hívódnak meg az IDT-n keresztül. Ez nem minden régi program esetén használható, hiszen például a BIOS hívása esetén (INT 10h) valójában a 16. kivétel (koprocesszor hiba) keletkezne. Bár a megszakításkezelőket át lehet alakítani, hogy a veremben lévő VM bitet ellenőrizzék, és szükség esetén az eredeti 16 bites megszakításkezelőnek adják át a vezérlést.

Az emulálás szempontjából sokkal hasznosabb, ha a virtuális módú taszkoknál az IOPL 3-nál kisebb értékre van állítva. Ekkor minden virtuális üzemmódban végrehajtott megszakítás általános védelmi hibát okoz. A kivételkezelő megvizsgálhatja a hibát okozó utasítást, és ha egy INT n utasítást talál (pl. CDh, CCh opkódok), akkor emulálhatja a megszakítást.

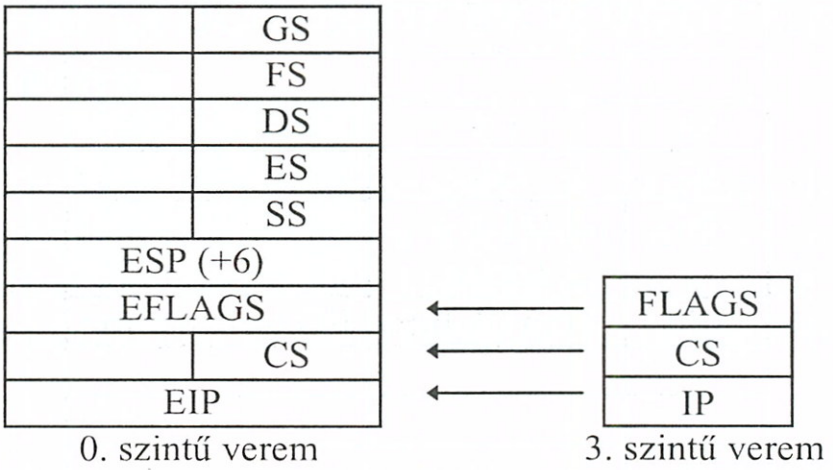
Az operandus által jelzett sorszámú megszakításnak a címe (CS:IP) megállapítható a 0000:0000-n lévő valós módú táblázat alapján. Ezt a címet kell a kivételkezelő vermében elhelyezni, hogy az IRETD hatására a virtuális módú megszakításra adódjon a vezérlés. A veremben lévő eredeti CS:IP címet (az INT utasítást követő cím) az EFLAGS régi értékével együtt a 3. szintű verembe kell rakni, és a 3 szintű veremmutatót csökkenteni kell 6-tal, mintha tényleg egy INT utasítás történt volna.



3.17. ábra A verem átalakítása az INT utasítás emulálásához

Amikor a virtuális módú megszakítás-kezelő végrehajtja az IRET utasítást, újra általános védelmi kivétel keletkezik. Ekkor a 13. kivételkezelőnek a 3. szintű veremből ki kell vennie a FLAGS és a CS:IP regisztereket, majd

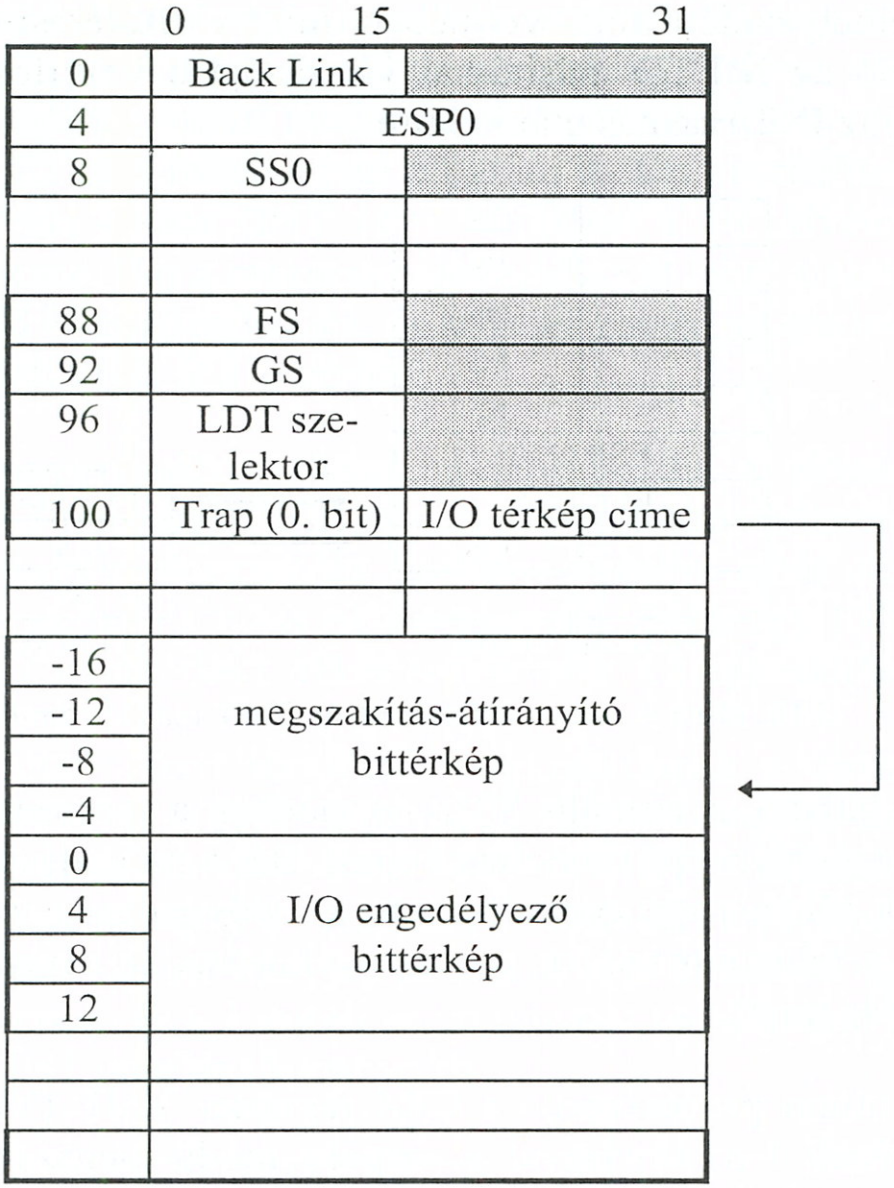
ezt kell elhelyeznie a saját vermében, mint visszatérési címet. Ezután a kivételkezelő az IRETD utasítással visszaadja a vezérlést a virtuális módú taszknak, az INT utasítás utáni címre.



3.18. ábra A verem átalakítása az IRET utasítás emulálásához

Mint látható, a virtuális 8086-os módban a megszakítások kezelése kicsit összetettebb a valós módhoz képest. Bár a működésüket tekintve a megszakítások tökéletesen emulálhatóak, az időveszteség mégis jelentős, elsősorban a privilégiumszintek gyakori változása, és a veremben lévő adatok cserélgetése miatt.

A Pentium processzoron a CR4 regiszter VME (Virtual Mode Extension) bitjének 1-re állításával a virtuális módú megszakítások kezelése leegyszerűsíthető. Ha a VME engedélyezve van, akkor a megszakításokat a megszakítás-átirányító bittérkép (Interrupt Redirection Bitmap) szabályozza. Ez a bittérkép az I/O engedélyező bittérképhez hasonlóan a TSS-ben helyezkedik el. Minden szoftver megszakításhoz egy bit tartozik, tehát a bittérkép mérete 32 byte (256/8). Ez a 32 byte közvetlenül az I/O bittérkép előtt foglal helyet, vagyis a TSS-ben lévő I/O báziscím jelzi a megszakítás-átirányítás bittérkép végének címét is. (Ld. 3.19. ábra) Ha egy bit magas, akkor az adott megszakítás esetén kivétel keletkezik az egyszerű virtuális módhoz hasonlóan. Viszont ha egy bit 0-ra van állítva, akkor a processzor nem hívja meg a felügyelőprogramot, hanem elvégzi a megszakítást anélkül, hogy kilépne a virtuális módból. Ezzel a kiterjesztéssel a virtuális módban futó programok sebességén jelentősen lehet javítani.



3.19. ábra A megszakítás-átirányító bittérkép elhelyezkedése

A virtuális üzemmód használatával tehát megoldódnak a kompatibilitási problémák, és szükség esetén védett módban is lehetőség nyílik a régi 16 bites alkalmazások futtatására. Azonban a legtöbb virtuális módot támogató operációs rendszer könnyen sebezhető, még akkor is, ha az IOPL<3. Akár néhány V86-os utasítással is súlyos károkat lehet okozni (pl. lefagyás), bár ez nem a virtuális mód, hanem inkább az adott operációs rendszer hibája.

4. A processzor felismerése

4.1. A 386-os processzor és a virtuális mód felismerése

Védett módba kapcsolás előtt célszerű azt megvizsgálni, hogy legalább 386-os, vagy jobb processzor van-e a gépben. A 286-os processzorokkal most nem foglalkozunk, mivel azok lényegében 16 bitesek, és a védett módból történő visszakapcsolás is elég körülményes (a legjobb módszer a triple fault).

Az alábbi rövid eljárás az EFLAGS-ben az NT (Nested Task) bitet használja a 386-os felismerésére, az ellenkezőjére állítja, és megnézi, hogy tényleg megváltozik-e. Ez a bit a 386-osnál jelent meg először, azelőtt fenntartott bit volt, és valós módban nem módosítható az értéke, a korábbi processzorokon viszont igen. Ha tehát megváltozik az NT bit, akkor egy hibaüzenettel kilép a program.

```
Check386 Proc

   pushf                ; az eredeti flag elmentése
   pushf                ; flag -> ax
   pop  ax
   xor  ah,40h          ; az NT bit megváltoztatása
   push ax              ; ax -> flag
   popf
   pushf                ; flag -> bx
   pop  bx
   popf ; az eredeti flag vissza
   cmp  ax,bx           ; megváltozott az NT bit?
   jne  NT_Error        ; igen, tehát nem 386-os.
   ret

NT_Error:
   lea  dx,No386Str
   jmp  Exit            ; hibaüzenet kiírása, kilépés.

Check386 Endp

No386Str db 'Ez a program csak 386 vagy jobb
         db 'processzoron fut.',10,13,'$'
```

Átkapcsolás előtt még azt is meg kell nézni, hogy egyáltalán valós módban vagyunk-e, hisz előfordulhat, hogy V86 módban fut a program, például Windows alatt egy DOS ablakban, vagy ha valamilyen memória manager aktív (Pl. EMM386, QEMM). Ha ilyenkor próbálunk védett módba kapcsolni, általános védelmi hiba jön létre. Ennek elkerülése érdekében egy hibaüzenettel kilépünk a programból, ha a CR0 regiszter PE bitje 1.

```
CheckV86 Proc

    smsw ax                 ; a CR0 alsó resze az ax-ba
    test al,1               ; a PE bit vizsgálata
    jnz  V86_Error          ; ha 1, védett módban vagyunk
    ret

V86_Error:
    lea  dx,V86Str
    jmp  Exit               ; kilépés hibaüzenettel.

CheckV86 Endp

V86Str  db 'Már védett módban vagyunk!',10,13,'$'
```

Itt nem használhatjuk a MOV EAX,CR0-t a PE bit vizsgálatára, mert ez privilegizált utasítás, és ha virtuális módban hajtjuk végre, akkor helytelen eredményt kapunk, általában az operációs rendszer az igazi CR0 helyett más értéket, pl. 0-t ad vissza. Az SMSW utasítás nem privilegizált, bár ezzel csak a CR0 regiszter alsó 16 bitjét kapjuk meg (a 66h prefix sem mindig hatásos), de a PE bit ellenőrzéséhez ez elegendő. A legtöbb védett módú program mindenesetre elköveti azt a hibát, hogy a virtuális üzemmódot a CR0 regiszter közvetlen elérésével vizsgálja meg.

4.2. A 486-os és Pentium processzorok felismerése

Ha egy programban olyan utasítások is szerepelnek, amelyek csak 486-oson, vagy csak Pentiumon léteznek (pl. XADD, CMPXCHG8B), akkor szükség van ezek felismerésére is. A 486-os processzor az AC bit (Alignment Check) segítségével ismerhető fel, mivel itt jelent meg először az illeszkedés ellenőrzés. A módszer ugyanaz, mint a 386-os esetén, megnézzük, hogy megváltoztatható-e a 18. bit az EFLAGS-ban.

```
Check486 Proc

    pushfd                  ; az eredeti eflag elmentése
    pushfd                  ; eflag -> eax
    pop  eax
    btc  eax,18             ; az AC bit megváltoztatása
    push eax                ; eax -> eflag
    popfd
    pushfd                  ; eflag -> ebx
    pop  ebx
    popfd                   ; az eredeti eflag vissza
    cmp  eax,ebx            ; megváltozott a bit?
    jne  AC_Error           ; igen, tehát nem 486-os.
    ret

AC_Error:
    lea  dx,No486Str
    jmp  Exit               ; hibaüzenet kiírása, kilépés.

Check486 Endp

No486Str db 'Ez a program csak 486 vagy jobb
         db 'processzoron fut.',10,13,'$'
```

Arra vigyázni kell, hogy a fentieket csak a 386-os ellenőrzése után szabad végrehajtani, mert a PUSHFD, POPFD, és a BTC nem létezik a korábbi processzorokon, ezért érvénytelen utasításnak minősül.

A Pentium és néhány 486-os esetén a CPUID utasítással határozható meg a processzor pontos típusa és bizonyos tulajdonságai. Az utasítás használata előtt azonban ellenőrizni kell az ID bitet az EFLAGS-ben, hasonlóan a fenti Check486-hoz, viszont a 18. bit helyett a 21. bitet kell megváltoztatni.

Sajnos, néhány processzornál nem ilyen egyszerű a helyzet. A NexGen processzoroknál például akkor is működhet a CPUID, ha az ID nem változtatható meg, és emiatt a felismerés nem sikerülhet. Erre egy megoldás, ha a 6. megszakítást (érvénytelen utasítás kivétel) átirányítjuk egy saját kezelőre, és végrehajtjuk a CPUID utasítást. Ha 6. kivétel keletkezik, akkor a processzor nem támogatja ezt az utasítást, ellenkező esetben további információk szerzésére használhatjuk a CPUID-et.

A Cyrix processzoroknál sem működik helyesen az ID bit, és emiatt sok program rosszul ismeri fel. A CPUID utasítás engedélyezéséhez két lépés szükséges. Először egy kis trükkel ellenőrizni kell a Cyrix processzor jelen-

létét. Ez azon alapul, hogy az alábbi osztás végrehajtása után a carry flag értéke 0 a Cyrixnál, és 1 a többi processzornál.

```
xor  dx,dx
mov  ax,5555h
mov  cx,2
clc
div  cx
```

Ha a teszt sikerült, akkor használhatjuk a CCR regisztereket. Ezek olyan speciális regiszterek, amelyet néhány processzorgyártó (Texas Instruments, Cyrix) használ a chipek bizonyos tulajdonságainak engedélyezésére, illetve letiltására. Ezek a regiszterek a processzoron belül vannak, és a 22h és a 23h portokon keresztül érhetők el, a 22h-ra kell kiírni a regiszter indexét, és a 23h porton keresztül írható/olvasható az adat. A CPUID utasítást a CCR4 (index E8h) 7. bitje határozza meg, ha ez 0, le van tiltva, ha pedig 1, akkor engedélyezve van. Az alábbi néhány utasítás után már a Cyrix processzorokon is alkalmazható a CPUID.

```
mov  al,0e8h          ; CCR4 index
out  22h,al
mov  ah,al
in   al,23h           ; a regiszter beolvasása
bts  ax,7             ; a CPUID engedélyezése
xchg al,ah
out  22h,al
mov  al,ah
out  23h,al           ; a regiszter kiírása
```

4.3. A CPUID utasítás használata

A 386-ostól kezdve a RESET után a DX-ben tartalmazták a processzorok az azonosítójukat. Korábban, ha egy program tudni akarta a processzor típusát, a 0040h:0067h megfelelő módosítása után egy RESET-et kellett végrehajtania a billentyűzet vezérlő segítségével (out 60h,0feh), vagy egy tripla hibát (triple fault) kellett előidéznie (pl. az IDT határ 0-ra állításával). Később inkább a flag bitek alapján határozták meg a processzort, mivel a RESET elég lassú módszer. (Bár a Pentiumokon 2 órajel alatt is végrehajtható a RESET, a cache törlése nélkül.)

A CPUID használatával tehát könnyedén megállapítható a processzor típusa és családja, de ezenkívül egyéb infomációkat is nyújt pl. a beépített cache-ről, a támogatott tulajdonságokról. Ha egy program valamilyen speciális funkciót használ (pl. VME, MMX), akkor előtte ellenőriznie kell, hogy az támogatva van-e az adott processzoron.

A CPUID utasítás (az opkódja 0Fh, 0A2h) az EAX tartalmától függően változtatja meg a 32 bites általános regiszterek értékét.

Az EAX = 0 esetén kaphatjuk meg a támogatott lekérdezések számát. Ez jelenleg 2, de az új funkciók megjelenésével fokozatosan növekszik.

Bemenet	EAX = 0
Kimenet	EAX = maximális bemeneti érték (jelenleg 1 vagy 2) EBX:EDX:ECX = gyártó azonosító string (12 karakter)

Gyártó azonosító stringek	
„GenuineIntel"	Intel 486, Pentium vagy Pentium Pro
„AuthenticAMD"	AMD 486DX2/DX4, 5x86, K5 vagy K6
„CyrixInstead"	Cyrix 5x86 vagy 6x86
„NexGenDriven"	NexGen 5x86 vagy 6x86
„UMC UMC UMC"	UMC U5S vagy U5D

Ha a CPUID utasítást az EAX = 1 értékkel hajtjuk végre, akkor megkaphatjuk a processzor pontos típusát, és a támogatott kiterjesztéseket.

Bemenet	EAX = 1
Kimenet	EAX = a processzor típusa EDX = tulajdonságok

Processzor típus, család, modell, reviziószám (EAX)		
Típus 12–13.bitek	11b	fenntartva
	10b	másodlagos dual processzor
	01b	OverDrive processzor
	00b	egy processzor (esetleg elsődleges dual)
Család 8–11.bitek	4	Intel 486, Cx5x86
	5	Intel 586, Cx6x86, AMD K5 és K6
	6	Intel 686

Modell 4–7. bitek	Intel 80486	0	486 DX-25/33
		1	486 DX-50
		2	486 SX
		3	486 DX2
		4	486 SL
		5	486 SX/2
		6	486 DX/2 WB
		7	486 DX/4
		8	486 DX/4 WB
	Intel Pentium	0	P5 A
		1	P5
		2	P54C
		3	P24T OverDrive
		4	P55C vagy P54CTB
		7	P54C
	Intel Pentium Pro	0	P6 A-step
		1	P6
		3	Klamath
	AMD 486	3	486 DX/2
		7	486 DX/2 WB
		8	486 DX/4
		9	486 DX/4 WB
		E	5x86
		F	5x86 WB
	AMD K5	0	SSA5 (PR75, PR90, PR100)
		1	5k86 (PR120, PR133)
		2	5k86 (PR166)
		3	5k86 (PR200)
	AMD K6	6	K6
	Cyrix 5x86	9	5x86
	Cyrix 6x86	2,3	6x86
	UMC 486	1	U5D
		2	U5S
	NexGen Nx586	0	Nx586 vagy Nx586FPU
0–3 bitek	revíziószám		

Az EDX-ben tárolt bitek jelzik, hogy bizonyos lehetőségek elérhetőek-e az ádott processzoron.

Jellemzők (EDX)	
bitek	tulajdonságok (ha az adott bit 1, támogatva van)
24–31.	fenntartva
23.	MMX utasítások
16–22.	fenntartva
15.	CMOVcc utasítások
14.	Machine Check Architecture
13.	PGE (Page Global Extension)
12.	MTRR (Memory Type Range Registers)
10–11.	fenntartva
9.	APIC
8.	CMPXCHG8B utasítás
7.	MCE (Machine Check Extension)
6.	PAE (Physical Address Extension)
5.	MSR (Model Specific Registers)
4.	TSC (Time Stamp Counter)
3.	PSE (Page Size Extension)
2.	DE (Debugging Extension)
1.	VME (Virtual Mode Extensions)
0.	a processzor FPU-t tartalmaz

Bemenet	EAX = 2
Kimenet	A 4 általános regiszter tartalmazza a cache konfigurációt (TLB, L1, L2)

II. RÉSZ

1. Egyszerű védett módú programok

1.1. Átkapcsolás védett módba

Ebben a fejezetben a lehető legegyszerűbb példa szerepel, átkapcsolás védett módba, és rögtön vissza.

A globális deszkriptortáblázatban most mindössze 2 deszkriptor lesz, egy 32 bites kódszegmens a védett módú kód számára, és egy 16 bites kódszegmens a valós módú visszatéréshez. Mivel most nem használunk megszakításokat sem, ezért az IDT-t sem szükséges inicializálni, ehhez viszont a megszakításoknak végig letiltva kell lenniük.

A védett módba kapcsoláshoz csupán a CR0 regiszter PE bitjét kell egyre állítani, de ez előtt még több fontos dolgot is el kell végezni. Az előző fejezet alapján biztosítható, hogy a processzor legalább 386-os legyen, illetve hogy a program valós módban fusson. Ha ezek a feltételek nem teljesülnek, akkor egy hibaüzenettel ki kell lépni, vagy ha virtuális módban vagyunk, akkor esetleg a védett módba kapcsoláshoz más utat is kereshetünk (pl. DPMI, VCPI). Az átkapcsolás előtt a GDT-t is mindenképpen be kell állítani, hogy érvényes deszkriptorokat tartalmazzon.

1.1.1. A globális deszkriptortáblázat beállítása

Első lépésként a GDT-ben létre kell hozni legalább két deszkriptort a kódszegmensek számára. A deszkriptortáblázatok felépítésére sokféle módszer létezik, leggyakrabban struktúrákat használnak, de a kisebb táblázatok akár közvetlenül „kézzel" is feltölthetőek.

A könyvben szereplő példákban a GDT-t az alábbi makró segítségével definiáljuk.

```
Descriptor Macro Name,Base,Limit,Type
    Name Equ $-GDT
    dw   Limit and 0ffffh
    dw   Base and 0ffffh
    db   (Base shr 16) and 0ffh
    dw   Type or ((Limit shr 8) and 0f00h)
    db   Base shr 24
Endm
```

Ennek első paramétere egy szimbólumnév, amelynek az értéke a GDT címétől való távolság lesz. Ez azért hasznos, mivel ez a szimbólum egyben egy szelektor is, vagyis ezt az értéket egy szegmensregiszterbe töltve egy érvényes szegmenset kapunk. Ez amiatt van, mert egy deszkriptor 8 byte-os, vagyis ezek a szimbólumok 8 többszörösei lesznek, a szelektorokban pedig az index mező a 2–15. biteken van, azaz 3 bittel eltolva, ami ugyancsak 8 többszöröse lesz. Például a GDT-ben szereplő 5. deszkriptorhoz tartozó szimbólum értéke 40 lesz, az ehhez tartozó 0. szintű (RPL = 0) GDT-re mutató (TI = 0) szelektor értéke pedig 5 shr 3, tehát ugyancsak 40.

Ezután következnek a deszkriptort meghatározó paraméterek: a bázis, a határ, és a típus mezők. Ez a makró éppen azért előnyös a struktúrával szemben, mert a deszkriptorokban az egyes mezők szétszabdalva, „összevissza" helyezkednek el. Ez a makró elvégzi ezt a szétdarabolást, és ezáltal az egész deszkriptortáblázat sokkal áttekinthetőbbé válik.

Az előző makrót felhasználva a globális deszkriptortáblázat a két kódszegmenssel együtt a következőképpen néz ki:

```
GDT Label Qword

Descriptor NullSel,0,0,0
Descriptor Code32,0,0ffffh,409ah
Descriptor Code16,0,0ffffh,9ah

GDT_Len  Equ $-GDT              ; a GDT hossza
GDT_Num  Equ GDT_Len/8-1        ; a deszkriptorok száma
```

A GDT első deszkriptora az ún. nulla deszkriptor, amely kihasználatlan, majd ezt követi két darab 0 báziscímű és 64 KB nagyságú kódszegmens. Az elérési jog mező értéke az alábbiakból adódik:

P	DPL		S	Típus			
1	0	0	1	1	0	1	0

1.1. ábra A kódszegmens deszkriptor elérési jogai

Vagyis ez az érték (10011010b = 9ah) a memóriában jelenlévő (present), 0. privilégiumszintű, végrehajtható, és olvasható kódszegmenset jelöl. Ehhez még hozzáadódik a Code32 esetén a 4000h, ami jelzi, hogy a D bit 1, tehát hogy ez egy 32 bites szegmens. A Code16 pedig egy 16 bites kódszegmens szelektora, mivel a deszkriptor D bitje 0.

A 32 bites kódszegmens mérete az egyszerűség kedvéért most 64 KB, ami jóval meghaladja a szegmens tényleges méretét, de most ez is megfelel. A szegmens akár 4 GB-os is lehetne, ehhez a deszkriptor típusát 0C09Ah-ra (G bit = 1), a határát pedig FFFFFh-ra kell állítani.

Az itt szereplő bázis csak relatív érték, hisz előre nem tudjuk, hogy a program pontosan melyik memóriacímen helyezkedik el, tehát ezt majd a program futása közben kell fizikai címmé alakítani. A GDT_Num tartalmazza a módosítani kívánt deszkriptorok számát (a nulla deszkriptor nélkül).

A deszkriptorok beállításán kívül a GDTR-t is inicializálni kell. Az LGDT utasításnak egy 48 bites memóriaoperandust kell megadni, amely tartalmazza a GDT hosszát (16 bit) és a báziscímét (32 bit). Mivel ez a báziscím is fizikai cím, ezért itt ugyancsak szükség van a kiigazításra a program aktuális helyétől függően.

```
GDT_Value                    Label Pword        ; az LDTR pa-
ramétere
GDT_Limit                    dw GDT_Len         ; a GDT hossza
GDT_Basedd offset GDT             ; a GDT címe
```

Az alábbi programrészlet a báziscímek átalakítását elvégzi, hozzáadva a program címét a deszkriptorok báziscím mezőjéhez, valamint beállítja a GDTR-t is.

```
InitGDT Proc

    mov  eax,ds           ; eax = adatszegmens
    shl  eax,4            ; eax = báziscím
    add  GDT_Base,eax     ; a GDT_Base fizikai címmé
         ; alakítása

    mov  cx,GDT_Num       ; a használt szelektorok száma
    mov  edx,offset GDT   ; ds:edx a GDT-re mutat
    add  edx,8            ; a nulla deszkriptor átugrása

    mov  bx,ax            ; bx = a Bázis alsó 16 bitje
```

```
    shr  eax,16             ; ax = a Bázis felsõ 16 bitje

Put_Base_Loop:
    add  [edx+2],bx         ; a Bázis  0-15 bitek tárolása
    add  [edx+4],al         ; a Bázis 16-23 bitek tárolása
    add  [edx+7],ah         ; a Bázis 24-31 bitek tárolása
    add  edx,8              ; lépés a következõre
    loop Put_Base_Loop      ; a többi deszkriptor...

    lgdt GDT_Value          ; a GDTR beállítása

    ret

InitGDT Endp
```

A 32 bites regiszterek természetesen valós módban is használhatóak, sőt elvileg a mov eax,ds is érvényes, amely egyenértékű a xor eax,eax és a mov ax,ds utasításpárral. Bár az egyik operandus 32 bites, a másik 16 bites, a TASM helyesen lefordítja, és az utasítás is a kívánt módon működik, vagyis az EAX alsó 16 bitjébe kerül a DS, a felső 16 bit pedig törlődik.

Ha tudnánk előre, hogy a védett módú program hova fog betöltődni, akkor a fenti címátalakításokra nem lenne szükség. Ha feltételeznénk, hogy a program mindig a 9000:0000 címen helyezkedik el, akkor a deszkriptorok, illetve a GDTR báziscíméhez csak hozzá kellene adni a 90000h értéket.

1.1.2. A szegmensek közötti ugrások

A példákban gyakran szerepelnek szegmensek közötti távoli ugrások. Erre akkor van szükség, ha például egy 16 bites kódszegmensből egy 32 bites kódszegmensre akarunk ugrani, vagy fordítva, illetve ha a CS regiszterbe új értéket szeretnénk betölteni.

A valós és a védett mód közötti átkapcsolások után is szerepelnie kell egy ugrásnak, az előolvasó sor (prefetch queue) törlése miatt. Ha itt egy távoli ugrást helyezünk el, akkor egyúttal a CS-be egy védett módú szelektort tölthetünk be, és átadhatjuk a vezérlést a 32 bites kódszegmensnek is.

A legtöbb assembler a távoli ugrásokat is közeli ugrásokra fordítja le, és ezért néha kicsit nehéz elérni a tényleges szegmensek közötti ugrást. A NASM (Netwide Assembler) viszont megengedi a jmp far szelektor:ofszet típusú utasítást is, és ezt helyesen fordítja le.

Ahhoz, hogy TASM alatt is könnyedén tudjunk a szegmensek között ugrani, néhány makrót definiálunk. Az alábbi makrókban szereplő 0EAh a távoli ugrás (FAR JMP) gépi kódja, a 66h prefix pedig az utána következő utasítás operandus méretét változtatja meg. Paraméterként a kódszegmens szelektorát és az ofszetcímet kell megadni.

```
Jump32_16  Macro Sel,Off     ; 32 -> 16 bites ugrás
    db 66h
    db 0eah
    dw Off
    dw Sel
    Endm

Jump16_32  Macro Sel,Off     ; 16 -> 32 bites ugrás
    db 66h
    db 0eah
    dd offset Off
    dw Sel
    Endm

Jump32_32  Macro Sel,Off     ; 32 bites ugrás
    db 0eah
    dd offset Off
    dw Sel
    Endm

Jump16_16  Macro Sel,Off     ; 16 bites ugrás
    db 0eah
    dw offset Off
    dw Sel
    Endm
```

Hasonló módon létrehozhatóak makrók a távoli eljáráshívások (FAR CALL) számára is, ennek gépi kódja a 09Ah.

1.1.3. Átkapcsolás védett módba és vissza

Most már minden feltétel adott a védett módba történő átkapcsoláshoz, csak a korábbi eljárásokat és makrókat kell használni. A példák különböző részei a könnyebb áttekinthetőség kedvéért külön file-okba kerülnek, a processzor ellenőrzése a check.inc-be, a GDT-vel kapcsolatos dolgok a gdt.inc-be, a szegmensek közötti ugrások pedig a jump.inc-be.

Az első példában három szegmens szerepel, amelyek a Pgroup nevű csoportot alkotják:

- RealSeg 16 bites valós módú szegmens
- ProtSeg 32 bites védett módú szegmens
- StackSeg veremszegmens a valós mód számára

A relatív címek nem az egyes szegmensek, hanem a Pgroup csoport kezdetéhez igazodik. Ez a későbbiekben is így lesz, tehát az ofszetcímek a valós és a védett módban egyaránt a Pgrouphoz képest lesznek relatívak.

Most a védett módba kapcsolás után átugrunk a 32 bites védett módú szegmensre, ahol egyszerűen csak visszaugrunk a 16 bites kódszegmensre a Code16 szelektor segítségével, majd visszakapcsolunk valós módba.

```
.386p    ; védett módú utasítások

include jump.inc          ; távoli ugrások definíciója

Pgroup  Group RealSeg,ProtSeg,StackSeg

RealSeg Segment Para Public Use16
        Assume  cs:Pgroup, ds:Pgroup

Real_Start:

   cli  ; megszakítások letiltása

   push cs                ; ds = Pgroup
   pop  ds

   call Check386          ; 386-os ellenõrzése
   call CheckV86          ; V86 mód ellenõrzése

   lea  dx,Prot_Str       ; átkapcsolás elõtti szöveg
   mov  ah,9              ; kiírása
   int  21h

   call InitGDT           ; GDT beállítása

   mov  eax,CR0           ; átkapcsolás védett módba
   or   al,1
   mov  CR0,eax           ; PE bit=1

   Jump16_32 Code32,Prot_Start
        ; ugrás a 32 bites kódszegmensre

Real_Back:

   mov  eax,CR0           ; visszakapcsolás valós módba
   and  al,not 1
```

```
    mov  CR0,eax          ; PE bit=0

    Jump16_16 Pgroup,Real_Back2
         ; a védett módú szelektor helyett
         ; valós módú szegmens a CS-be

Real_Back2:

    lea  dx,Real_Str

Exit:
    mov  ah,9
    int  21h              ; string kiírása
    mov  ah,4ch
    int  21h              ; kilépés

Prot_Str db 'Példa 1: átkapcsolás védett módba...$'
Real_Str db 'és vissza.',10,13,'$'

RealSeg Ends


ProtSeg Segment Para Public Use32

Prot_Start:

    nop  ; 32 bites „semmittevés"

    Jump32_16 Code16,Real_Back
         ; visszaugrás a 16 bites szegmensre
         ; ahol visszakapcsolunk valós módba

ProtSeg Ends


StackSeg Segment Stack Use16; 1 KB valós módú verem
db  400h dup (?)
StackSeg Ends

include check.inc         ; Check386 és CheckV86
include gdt.inc           ; GDT és InitGDT

End Real_Start
```

Ez a példaprogram csak az alapvető elemeket tartalmazza, hisz most védett módban nincs adat-, és veremszegmens definiálva, és a megszakítások sincsenek inicializálva. Ezekről majd a következő példákban lesz szó.

1.2. Adatszegmens elérése

Ahhoz, hogy védett módban a ProtSeg-et elérjük adatszegmensként, csak be kell illeszteni a GDT-be a megfelelő adatdeszkriptort:

```
GDT Label Byte

Descriptor NullSel,0,0,0
Descriptor Code32,0,0ffffh,409ah
Descriptor Code16,0,0ffffh,9ah
Descriptor Data32,0,0ffffh,4092h
Descriptor Data16,0,0ffffh,92h

GDT_Len  Equ $-GDT              ; a GDT hossza
GDT_Num  Equ GDT_Len/8-1        ; a deszkriptorok száma
```

Mint látható, a Data32 és a Data16 is 0 bázisú, 64 KB nagyságú szegmens. A 92h egy jelenlévő, 0. privilégiumszintű, írható és olvasható adatszegmenset jelöl:

P	DPL		S	Típus			
1	0	0	1	0	0	1	0

1.2. ábra Az adatszegmens elérési jogai

A Data32-nél a B bit értéke 1, ami jelzi, hogy ez egy 32 bites adatszegmens, és emiatt a 92h-hoz hozzá kell adni 4000h-t.

Az InitGDT eljárás az adatszegmensek báziscímét is automatikusan módosítja a program aktuális címétől függően, tehát a makróban szereplő 0 báziscím nem fizikai cím, hanem a Pgroup csoporthoz viszonyított cím.

Az üzemmódváltások során a szegmensregisztereket három alkalommal kell új értékkel feltölteni. Először a védett módba kapcsolás után – de még a 32 bites kódszegmensre való ugrás előtt – a szegmensregiszterekbe betöltjük a 32 bites adatszegmens szelektorát (Data32).

A valós módba kapcsolás előtt minden szegmensregiszterben olyan deszkriptornak kell lennie, amelynek határa a valós módnak megfelelően 64 KB. Ez akkor igazán fontos, ha védett módban a szegmensregiszterekben más határral (pl. 4 GB) rendelkező deszkriptorok vannak.

Végül pedig a visszakapcsolás után a szegmensregiszterekben valós módú szegmens értéket kell majd elhelyezni.

A 32 bites adatszegmens kipróbálásának legegyszerűbb módja, ha elhelyezünk a szegmensben egy statikus adatot, majd ezt védett módban betöltjük az egyik regiszterbe, utána pedig valós módban megnézzük, hogy megegyezik-e a két érték, és az eredményt kiírjuk a képernyőre.

```
.386p    ; védett módú utasítások használata

include jump.inc          ; távoli ugrások definíciója

Pgroup  Group RealSeg,ProtSeg,StackSeg

RealSeg Segment Para Public Use16
        Assume  cs:Pgroup, ds:Pgroup

Real_Start:
   cli  ; megszakítások letiltása

   push cs                ; ds = Pgroup
   pop  ds

   call Check386          ; 386-os ellenõrzése
   call CheckV86          ; V86 mód ellenõrzése

   lea  dx,Test_Str       ; a teszt szöveg kiírása
   mov  ah,9
   int  21h

   call InitGDT           ; GDT beállítása

   mov  eax,CR0           ; átkapcsolás védett módba
   or   al,1
   mov  CR0,eax           ; PE bit=1

   mov  ax,Data32         ; védett módú adatszegmens-
   mov  ds,ax             ; szelektor betöltése a
   mov  es,ax             ; szegmensregiszterekbe
   mov  fs,ax
   mov  gs,ax

   Jump16_32 Code32,Prot_Start
        ; ugrás a 32 bites kódszegmensre

Real_Back:

   mov  ax,Data16         ; 16 bites adatszegmens-
   mov  ds,ax             ; szelektor betöltése a
```

```
    mov  es,ax              ; szegmensregiszterekbe
    mov  fs,ax
    mov  gs,ax

    mov  eax,CR0            ; visszakapcsolás valós módba
    and  al,not 1
    mov  CR0,eax            ; PE bit=0
    jmp  $+2

    mov  ax,Pgroup          ; valós módú érték a
    mov  ds,ax              ; szegmensregiszterekbe
    mov  es,ax
    mov  fs,ax
    mov  gs,ax

    Jump16_16 Pgroup,Real_Back2
         ; a védett módú szelektor helyett
         ; valós módú szegmens a CS-be

Real_Back2:

    lea  dx,Success         ; feltételezzük a sikert
    cmp  ebx,12345678h      ; az ebx ellenõrzése
    je   Exit               ; sikerült

    lea  dx,Failed          ; nem sikerült

Exit:
    mov  ah,9
    int  21h                ; string kiírása
    mov  ah,4ch
    int  21h                ; kilépés

Test_Strdb 'Példa 2: a 32 bites adatszegmensben '
         db 'lévõ érték tesztelése $'
Success db 'sikerült.',10,13,'$'
Failed  db 'NEM sikerült.',10,13,'$'

RealSeg  Ends


ProtSeg  Segment Para Public Use32

Prot_Start:

    mov  ebx,Test           ; teszt érték az ebx-be

    Jump32_16 Code16,Real_Back
         ; visszaugrás a 16 bites szegmensre
         ; ahol visszakapcsolunk valós módba

Test  dd 12345678h          ; próba érték

ProtSeg  Ends
```

```
StackSeg Segment Stack Use16
db 400h dup (?)
StackSeg Ends

include check.inc          ; Check386 és CheckV86
include gdt.inc            ; GDT és InitGDT

End   Real_Start
```

1.3. A veremszegmens

Egy védett módú verem létrehozásához mindössze egy GDT-ben lévő deszkriptor (Stack32) és egy veremterület (Stack32Seg) szükséges.

```
GDT Label Byte

Descriptor NullSel,0,0,0
Descriptor Code32,0,0ffffh,409ah
Descriptor Code16,0,0ffffh,9ah
Descriptor Data32,0,0ffffh,4092h
Descriptor Data16,0,0ffffh,92h
Descriptor Stack32,0,0ffffh,4092h

GDT_Len  Equ $-GDT              ; a GDT hossza
GDT_Num  Equ GDT_Len/8-1        ; a deszkriptorok száma
```

A veremszegmens deszkriptora valójában egy egyszerű adatszegmens, a báziscíme, a határa, és a típusa megegyezik a Data32-vel, a Stack32 helyett akár a Data32-t is be lehetne tölteni az SS-be.

A mostani megoldás nem igazán biztonságos, de kis veremhasználatnál még elfogadható. Később, amikor már képesek leszünk a kivételek kezelésére, megakadályozható az is, hogy a verem méretének túllépésével ne írjunk felül más memóriaterületet. A lefelé bővülő veremszegmens használatával még biztonságosabbá tehető a verem, sőt túlcsordulásnál akár a verem mérete is megnövelhető. Ez az éppen futó program számára szinte észrevehetetlen, vagyis a verem nagyságának megváltoztatásával a program tovább folytatható, mintha nem is lépte volna túl a verem maximális méretét.

A védett módú verem inicializálásához tehát a verem szelektorát be kell tölteni az SS-be, a verem tetejének ofszetcímét pedig az ESP-be. Valós

módba való visszatéréskor pedig az SS:SP-t vissza kell állítani a korábbi valós módú veremre.

A verem ellenőrzése most egyszerűen néhány push/pop utasításpárral elvégezhető.

```
.386p

include jump.inc

Pgroup  Group RealSeg,ProtSeg,StackSeg

RealSeg Segment Para Public Use16
        Assume  cs:Pgroup, ds:Pgroup

Real_Start:
   cli  ; megszakítások letiltása

   push cs                ; ds = Pgroup
   pop  ds

   call Check386          ; 386-os ellenõrzése
   call CheckV86          ; V86 mód ellenõrzése

   lea  dx,Ex3_Str        ; a teszt szöveg kiírása
   mov  ah,9
   int  21h

   call InitGDT           ; GDT beállítása

   mov  eax,CR0           ; átkapcsolás védett módba
   or   al,1
   mov  CR0,eax           ; PE bit=1
   jmp  $+2

   mov  ax,Data32         ; védett módú adatszegmens-
   mov  ds,ax             ; szelektor betöltése a
   mov  es,ax             ; szegmensregiszterekbe
   mov  fs,ax
   mov  gs,ax

   mov  ax,Stack32        ; védett módú verem beállítása
   mov  ss,ax
   lea  esp,Pgroup:Stack32_End

   Jump16_32 Code32,Prot_Start
        ; ugrás a 32 bites kódszegmensre
```

```
Real_Back:

    mov  ax,Data16        ; 16 bites adatszegmens-
    mov  ds,ax            ; szelektor betöltése a
    mov  es,ax            ; szegmensregiszterekbe
    mov  fs,ax
    mov  gs,ax
    mov  ss,ax

    mov  eax,CR0          ; visszakapcsolás valós módba
    and  al,not 1
    mov  CR0,eax          ; PE bit=0

    mov  ax,Pgroup        ; valós módú érték a
    mov  ds,ax            ; szegmensregiszterekbe
    mov  es,ax
    mov  fs,ax
    mov  gs,ax

    mov  ax,StackSeg      ; valós módú veremszegmens
    mov  ss,ax            ; betöltése az SS-be
    mov  sp,400h          ; SP = a veremszegmens teteje

    Jump16_16 Pgroup,Real_Back2
         ; a védett módú szelektor helyett
         ; valós módú szegmens a CS-be

Real_Back2:

    lea  dx,Success       ; ha ide eljutottunk,
         ; akkor a verem mûködik.

Exit:
    mov  ah,9
    int  21h              ; string kiírása
    mov  ah,4ch
    int  21h              ; kilépés

Ex3_Str  db 'Példa 3: a 32 bites veremszegmens '
         db 'használata $'
Success  db 'sikerült.',10,13,'$'
Failed   db 'NEM sikerült.',10,13,'$'

RealSeg  Ends


ProtSeg  Segment Para Public Use32

Prot_Start:

    call Stack_Test       ; a verem tesztelése
```

```
    Jump32_16 Code16,Real_Back
         ; visszaugrás a 16 bites szegmensre

Stack_Test:

    push eax              ; néhány regiszter elmentése
    push edi
    push ds

    nop

    pop  ds
    pop  edi
    pop  eax

    ret  ; a call/ret is használja a
         ; vermet.

ProtSeg  Ends


Stack32Seg Segment Use32      ; védett módú verem
db 400h dup (?)
Stack32_End Label Byte
Stack32Seg Ends


StackSeg Segment Stack Use16 ; valós módú verem.
db 400h dup (?)
StackSeg Ends

include check.inc             ; Check386 és CheckV86
include gdt.inc               ; GDT és InitGDT

EndReal_Start
```

1.4. A képernyőszegmens elérése

A következő kis példa védett módban teleírja a szöveges képernyőt 'A' betűkkel. Ehhez kell egy 32 bites adatszegmens deszkriptor a GDT-ben, amely a videomemóriára mutat, ennek bázisa a 0b80000h fizikai cím (mono monitor esetén 0b0000h), és határa 80*25*2, azaz 4000 byte.

```
GDT Label Byte

Descriptor NullSel,0,0,0
Descriptor Code32,0,0ffffh,409ah
Descriptor Code16,0,0ffffh,9ah
```

```
Descriptor Data32,0,0ffffh,4092h
Descriptor Data16,0,0ffffh,92h
Descriptor Stack32,0,0ffffh,4092h
Descriptor Screen,0b80000h,4000,4092h

GDT_Len  Equ $-GDT              ; a GDT hossza
GDT_Num  Equ GDT_Len/8-2        ; a deszkriptorok száma
```

Arra vigyázni kell, hogy az itt lévő bázis már fizikai cím, ezért nem kell a program bázisát hozzáadni, mint pl. a kód-, vagy az adatszegmensek esetében. Ezért a GDT_Num-ból le kell vonni a nulla deszkriptoron kívül a képernyő deszkriptorát is.

A képernyő feltöltése a rep stosd utasítással gyorsan megoldható, ha az ES:EDI a képernyő kezdetére mutat, és az ECX-ben a képernyőn lévő dupla szavak száma van, az EAX pedig két 'A' betűt, és azok attribútumát tartalmazza.

```
ProtSeg Segment Para Public Use32

Prot_Start:

   push es                      ; eredeti ES elmentése

   mov  ax,Screen               ; ES:EDI = a videomemória
   mov  es,ax
   xor  edi,edi

   mov  ecx,80*25*2/4           ; 80*25 darab
   mov  eax,07410741h           ; szürke színû A betûk
                                ; (41h='A', 07h=szin)
   rep  stosd                   ; kiírása a képernyõre

   pop  es                      ; ES visszaállítása

   Jump32_16 Code16,Real_Back
                        ; visszaugrás a 16 bites szegmensre

ProtSeg Ends
```

Hasonló módon létrehozható egy deszkriptor, amely a grafikus videomemóriára mutat, ezáltal a megfelelő mód beállítása után nemcsak a szöveges, hanem a grafikus képernyőre is írhatunk.

1.5. A kivételek kezelése

Itt az ideje a kivételekkel is foglalkozni, hisz ahogy egyre összetettebbé válnak a programok, egyre több a hibalehetőség is.

Eddig, ha bármi hiba fordult volna elő a programokban, az a gép „lefagyását" okozta volna, hisz a védett módú megszakítások inicializálatlanok voltak, így az adott kivételkezelő nem hajtódhatott volna végre. A kivételek használatával a hiba oka, és pontos helye könnyen kideríthető, ezáltal könnyebben ki is javítható.

1.5.1. Az IDT beállítása

Az IDT definiálásához célszerű újra egy makrót használni. Paraméterként a megszakítás sorszámát, a kezelő eljárás szelektorát, és ofszetcímét kell megadni. A makró a sorszámnak megfelelő deszkriptort elhelyezi a táblázaton belül, és beállítja a megszakításkapu megfelelő mezőit.

```
Int_Gate MACRO Sel,Off
    org   IDT+Num*8
    dw    small Off
    dw    Sel
    db    0
    db    8eh
    dw    0
Endm
```

A makróban az ORG direktívával adjuk meg az adott deszkriptor helyét. Ennek előnye, hogy az egyes deszkriptorok bármilyen sorrendben követhetik egymást, illetve hogy a felesleges, nem használt kapukat így nem szükséges definiálni. Vagyis ha például csak a 0. és a 100. kaput akarjuk beállítani, akkor csak két makrónak kell szerepelnie az IDT-ben, a többi 98 pedig nyugodtan elhagyható. Mindössze az az egy megkötés van az ORG miatt, hogy a legmagasabb számú megszakítás a táblázat legvégén szerepeljen.

Az Int_Gate makró tehát létrehoz egy megszakításkaput (típus = 14), amely egy jelenlévő (P = 1), nulladik szintű (DPL = 0) rendszerdeszkriptor (S = 0).

P	DPL		S	Típus			
1	0	0	0	1	1	1	0

1.3. ábra A megszakításkapu elérési joga (8eh)

1.5.2. A kivételkezelők

A kivételkezelés legegyszerűbb módja, ha bármilyen hiba esetén egyszerűen visszakapcsolunk valós módba, és kilépünk a programból. Vagyis ekkor az IDT-ben létre kell hozni 16 (vagy több) kaput, amelyek a 32 bites kódszegmensben lévő kivételkezelőre mutatnak, ahol egy távoli ugrással majd visszakapcsolunk valós módba. Az ofszetcím megadásánál figyelni kell arra, hogy az ofszetcímek a teljes csoporthoz képest legyenek relatívak, és ne pedig a 32 bites szegmenshez képest (a Code32 báziscíme most a Pgroup kezdete).

```
IDT  Label QWord

Int_Gate 0,Code32,Pgroup:Exceptions
Int_Gate 1,Code32,Pgroup:Exceptions
Int_Gate 2,Code32,Pgroup:Exceptions
Int_Gate 3,Code32,Pgroup:Exceptions
Int_Gate 4,Code32,Pgroup:Exceptions
Int_Gate 5,Code32,Pgroup:Exceptions
Int_Gate 6,Code32,Pgroup:Exceptions
Int_Gate 7,Code32,Pgroup:Exceptions
Int_Gate 8,Code32,Pgroup:Exceptions
Int_Gate 9,Code32,Pgroup:Exceptions
Int_Gate 0ah,Code32,Pgroup:Exceptions
Int_Gate 0bh,Code32,Pgroup:Exceptions
Int_Gate 0ch,Code32,Pgroup:Exceptions
Int_Gate 0dh,Code32,Pgroup:Exceptions
Int_Gate 0eh,Code32,Pgroup:Exceptions
Int_Gate 0fh,Code32,Pgroup:Exceptions
Int_Gate 10h,Code32,Pgroup:Exceptions

IDT_Len Equ $-IDT
```

Mivel a kapudeszkriptorokban szelektorok szerepelnek, ezért a GDT-től eltérően nincs szükség az IDT deszkriptorok módosítására a program aktuális pozíciójától függően. Az IDTR beállításakor viszont már figyelembe kell venni a fizikai báziscímet.

Az IDTR-be a GDTR-hez hasonlóan egy 16 bites határt, és egy 32 bites báziscímet kell betölteni. A határt az IDT_Len címke tartalmazza, ez jelenleg 108 byte. Az IDT mérete mind a 256 kapudeszkriptor esetén 8 KB lenne, de természetesen az igényeknek megfelelően ennél kevesebb megszakítást is használhatunk. Most például elég 10h megszakítást definiálni, hisz egyelőre csak a kivételeket használjuk, és ezek közül is csak néhány fordulhat elő.

Amikor visszakapcsolunk valós módba, az IDTR-be a valós módú megszakításoknak megfelelő értéket kell betölteni, vagyis 03FFh határt (256 darab 4 byte-os vektor –1), és 0 báziscímet.

```
RealSeg  Segment

InitIDT  Proc

   mov   eax,cs
   shl   eax,4
   add   IDT_Base,eax          ; az IDT_Base fizikai
                               ; címmé alakítása
   lidt  IDT_Value             ; az IDTR beállítása

   ret

InitIDT   Endp

RestoreIDT Proc

   lidt Real_IDT_Value         ; a valós módú IDT
   ret

RestoreIDT Endp

IDT_Value Label  Pword         ; az IDTR paramétere
IDT_Limit dw  IDT_Len          ; 16 bites határ
IDT_Base  dd offset IDT        ; 32 bites báziscím

Real_IDT_Value Label Pword     ; IDTR a valós módban
Real_IDT_Limit dw 3ffh         ; 16 bites határ
Real_IDT_Base  dd 0            ; 32 bites báziscím

RealSeg  Ends
```

Védett módban most az összes kivétel az Exceptions eljárásra mutat, amely a 16 bites kódszegmensre ugrik, hogy visszakapcsoljon valós módba.

```
ProtSeg  Segment

Exceptions:

   Jump32_16 Code16,Real_Exc

ProtSeg  Ends
```

Ahhoz, hogy meg lehessen különböztetni a kivételeket és az egyszerű kilépést, a szokásos Real_Back helyett a Real_Exc-re ugrunk, és kiírjuk a képernyőre, hogy kivétel keletkezett.

```
.386p

include jump.inc

Pgroup  Group RealSeg,ProtSeg,StackSeg

RealSeg Segment Para Public Use16
        Assume  cs:Pgroup, ds:Pgroup

Real_Start:
   cli                          ; megszakítások letiltása

   push cs                      ; ds = Pgroup
   pop  ds

   call Check386                ; 386-os ellenõrzése
   call CheckV86                ; V86 mód ellenõrzése

   lea  dx,Ex5_Str              ; a teszt szöveg kiírása
   mov  ah,9
   int  21h

   call InitGDT                 ; GDT beállítása
   call InitIDT                 ; IDT beállítása

   mov  eax,CR0                 ; átkapcsolás védett módba
   or   al,1
   mov  CR0,eax                 ; PE bit=1
   jmp  $+2

   mov  ax,Data32               ; védett módú adatszegmens-
   mov  ds,ax                   ; szelektor betöltése a
   mov  es,ax                   ; szegmensregiszterekbe
   mov  fs,ax
   mov  gs,ax

   mov  ax,Stack32              ; védett módú verem beállítása
   mov  ss,ax
   lea  esp,Pgroup:Stack32_End

   Jump16_32 Code32,Prot_Start
                                ; ugrás a 32 bites kódszegmensre

Real_Exc:                       ; kivétel esetén ide ugrunk.

   mov  bl,1                    ; a kivétel jelzése a bl-ben.

   db   0b8h                    ; egy kis trükk

Real_Back:

   xor  bl,bl                   ; ha nincs kivétel, bl=0
```

```
    mov   ax,Data16             ; 16 bites adatszegmens-
    mov   ds,ax                 ; szelektor betöltése a
    mov   es,ax                 ; szegmensregiszterekbe
    mov   fs,ax
    mov   gs,ax
    mov   ss,ax

    mov   eax,CR0               ; visszakapcsolás valós módba
    and   al,not 1
    mov   CR0,eax               ; PE bit=0

    mov   ax,Pgroup             ; valós módú érték a
    mov   ds,ax                 ; szegmensregiszterekbe
    mov   es,ax
    mov   fs,ax
    mov   gs,ax

    mov   ax,StackSeg           ; valós módú veremszegmens
    mov   ss,ax                 ; betöltése az SS-be
    mov   sp,400h               ; SP = a veremszegmens teteje

    Jump16_16 Pgroup,Real_Back2
                                ; a védett módú szelektor helyett
                                ; valós módú szegmens a CS-be

Real_Back2:

    call RestoreIDT             ; valós módú megszakítások
                                ; visszaállítása

    lea   dx,Exc_Str
    or    bl,bl                 ; volt kivétel?
    jne   Exit                  ; igen.

    lea   dx,NoExc_Str          ; nem.

Exit:
    mov   ah,9
    int   21h                   ; string kiírása
    mov   ah,4ch
    int   21h                   ; kilépés

Ex5_Str    db 'Példa 5: kivételek kezelése...$'
NoExc_Str  db 'nem történt '
Exc_Str    db 'kivétel!',10,13,'$'

RealSeg Ends
```

A Real_Exc-nél szereplő B8h byte segítségével „elrejthető" a xor bl,bl utasítás, vagyis a bl-ben továbbra is 1 lesz, jelezve a kivételt.

Real_Exc		Real_Back	
mov bl,1 mov ax,0DB32h	B3, 01 B8, DB, 32	xor bl,bl	DB, 32

1.4. ábra A mov bl,1 és a xor bl,bl utasítások kódjai

A kivételkezelés működését például a nullával való osztással ellenőrizhetjük, aminek hatására a 0. kivétel keletkezik.

```
ProtSeg Segment Para Public Use32

Prot_Start:

    xor  bx,bx                  ; nullával való osztás
    div  bx                     ; 0. kivétel generálása

    Jump32_16 Code16,Real_Back
                                ; visszaugrás??

ProtSeg Ends
```

A fenti példa a legalapvetőbb kivételkezelés, bármilyen hiba esetén egyszerűen valós módba kapcsolunk, és kilépünk a programból. Ez egy kisebb programnál még elfogadható, viszont egy összetettebb rendszer esetén elengedhetetlen egy fejlettebb kivételkezelés használata.

Később, amikor már lesznek magasabb szintű képernyőkezelő eljárások, kivétel fellépése esetén a képernyőre kiírható a kivételt okozó utasítás pontos címe, és az egyes regiszterek aktuális tartalma. Ez nagyban megkönnyíti az esetlegesen előforduló hibák felismerését, és gyors kijavítását.

1.6. A megszakítások kezelése

1.6.1. A programozható megszakítás-vezérlő

Valós módban a 0–7. IRQ-k (Interrupt ReQuest) az INT 8-Fh-n vannak, ami ütközik a védett módú kivételek egy részével. Bár az Intel a 0–1Fh közötti megszakításokat fenntartotta a kivételek számára, sajnos mégis úgy alakult, hogy a két különböző megszakítástípus átfedi egymást. Mivel a kivételek vektora rögzített, ezért az IRQ-kat kellene más helyre átrakni, vagy

pedig a megszakítás-kezelőknek kellene felismerniük, hogy kivételként vagy hardver megszakításként hívódtak-e meg.

Mind a két megoldás megvalósítható a 8259A programozható megszakítás-vezérlő (Programmable Interrupt Controller, PIC) segítségével. Két PIC van: az elsődleges az IRQ 0–7 megszakításokat kezeli, és a 20h/21h port tartozik hozzá, a másodlagos PIC pedig az IRQ 8–15-t kezeli, és a 0a0h/0a1h portokon keresztül érhető el.

A megszakítás-vezérlőnek sok funkciója van, amelyek közül hármat érdemes kiemelni:

- Az IRQ-k helyének meghatározása
- Az egyes IRQ-k engedélyezése ill. letiltása
- A megszakítás-kezelő rutin végének jelzése. (EOI)

Az első funkciót használjuk majd fel, hogy a kivételek és az IRQ-k ne ütközzenek. A második azért hasznos, mert így szabályozható, hogy mely IRQ-k generálódhatnak, és melyek nem. A 21h ill. 0a1h portokon minden bit megfelel egy-egy IRQ-nak. Ha ez a bit 0, akkor az adott megszakítás engedélyezve van, ha 1, akkor le van tiltva.

Ha például csak az időzítés (IRQ 0) és a billentyűzet (IRQ 1) megszakításokat szeretnénk engedélyezni, a többit pedig letiltani (soros és párhuzamos portok kezelése, stb.), a 21h portra az FCh értéket kell kiírni.

Fontos még, hogy ha egy IRQ kezelő rutin befejeződött, akkor a PIC-nek egy EOI (End Of Interrupt) jelet kell küldeni, hogy a következő megszakítás is generálódhasson. Ugyanis mindaddig nem kerül kiszolgálásra új IRQ, amíg az előző nem végzett. A megszakítások végét lehet jelezni egyedileg (specific EOI), vagy általánosan (non-specific EOI) is, az utóbbi eset a gyakoribb.

1.6.2. Az IRQ-k áthelyezése

A hardvermegszakítások áthelyezéséhez a PIC megfelelő portjaira 4 vezérlőszót (Initialization Command Word) kell kiírni. Ezek 8 bites adatok, az első byte-ot (ICW1) a 20h, illetve az A0h portokra kell kiírni, a többi byte-ot (ICW2-ICW4) pedig a 21h, illetve az A1h portokra.

Az 1. byte 4. bitje jelzi, hogy ez az ICW1, a 0. bit pedig azt, hogy az ICW4-re is szükség van (bizonyos esetekben csak 3 byte kell). Az ICW2 a legfontosabb számunkra, mivel ezzel tudjuk elkerülni, hogy a megszakítások és a kivételek ütközzenek. Az ICW2 alsó 3 bit mindig 0, ezért a megszakítások báziscíme csak 8 többszöröse lehet. Az ICW3-al a két PIC összekötése vezérelhető (IRQ2, IRQ9), az ICW4-el pedig az EOI módja adható meg.

Az alábbi eljárás a PIC átprogramozásával a DH illetve a DL regiszterekben megadott értéknek megfelelően átállítja az IRQ-kat.

```
SetPIC Proc

    ; A 0-7. hardver megszakítások áthelyezése

    mov  al,10001b              ; ICW1 Bit 0: ICW4 is kell
    out  20h,al
    mov  al,dh                  ; ICW2: a master IRQ-k új
    out  21h,al                 ; báziscíme
    mov  al,100b                ; ICW3: master+slave: irq 2
    out  21h,al
    mov  al,1                   ; ICW4: Bit 0: 8086 mód
    out  21h,al                 ; Bit 1: megszakítás vége
                                ; (0=nem automatikus)

    ; A 8-15. hardver megszakítások áthelyezése

    mov  al,10001b              ; ICW1
    out  0a0h,al
    mov  al,dl                  ; ICW2: a slave IRQ-k új
    out  0a1h,al                ; báziscíme
    mov  al,10b                 ; ICW3: master+slave: irq 9
    out  0a1h,al
    mov  al,1                   ; ICW4
    out  0a1h,al

    ret

SetPIC Endp
```

Védett módba lépés előtt el kell menteni az engedélyezett megszakítások maszkját, be kell állítani az új értéket, és át kell helyezni az IRQ-kat. Az egyes megszakítások aktuális állapotát (engedélyezett vagy letiltott) megtudhatjuk a 21h illetve az A1h portról beolvasott byte-ból, és a megváltoztatott új maszkot is ezekre a portokra kell visszaírni. Valós módba kapcsoláskor ennek fordítottját kell tenni, vagyis visszaállítani a korábbi értékeket. A példánkban a szokásos 20h-2fh megszakításokon lesznek az IRQ-k.

```
RealSeg Segment

; IRQ-k átirányítása a 20h-2fh-ra

InitPIC Proc

   push ax dx

   in   al,21h
   mov  Old_Mask0,al          ; régi értékek elmentése
   in   al,0a1h
   mov  Old_Mask1,al

   mov  dx,2028h              ; IRQ -> int 20-2fh
   call SetPIC

   mov  al,0feh               ; csak az IRQ 0
   out  21h,al                ; engedélyezett
   mov  al,0ffh               ; a 8-15. megszakítások
   out  0a1h,al               ; letiltása

   pop  dx ax
   ret

InitPIC Endp

; IRQ-k visszaállítása a valós módnak megfelelõen.

RestorePIC Proc

   push ax dx

   mov  dx,0870h              ; IRQ -> int 08h, 70h
   call SetPIC

   mov  al,Old_Mask0          ; régi érték
   out  21h,al                ; visszaállítása
   mov  al,Old_Mask1
   out  0a1h,al

   pop  dx ax
   ret

RestorePIC Endp

Old_Mask0  db 0               ; elsõdleges PIC régi maszkja
Old_Mask1  db 0               ; másodlagos PIC régi maszkja

RealSeg Ends
```

1.6.3. IRQ 0 – időzítés

A mostani példában az IRQ 0-t fogjuk felhasználni, ami az időzítésre szolgál (timer interrupt), és alapesetben másodpercenként 18,2-szer generálódik (ez az érték persze megváltoztatható). A megszakítás-kezelő mindösszesze két dolgot fog csinálni: minden alkalommal megnöveli eggyel a számlálót, illetve a képernyő bal felső sarkában lévő karaktert. Ezt a számlálót használjuk majd fel időzítésre, azaz pontosan 5 másodpercig várakozunk, mialatt a képernyő sarkában lévő karakter folyamatosan változik.

Az IDT-ben létre kell hozni egy bejegyzést az IRQ 0 (vagyis INT 20h) számára, ami az időzítést kezelő megszakításra mutat.

```
Int_Gate 20h,Code32,Pgroup:IRQ0_Handler
```

Tehát az alábbi IRQ0_Handler folyamatosan megnöveli a számlálót, illetve a képernyő sarkában lévő karaktert. A képernyőre íráshoz a korábbi Screen szelektort használjuk fel. A megszakítás-kezelő által megváltoztatott regisztereket természetesen el kell menteni a veremben, és később vissza kell állítani. A megszakítás végét védett módban is a 20h portra kiírt 20h (EOI) értékkel kell jelezni, akárcsak valós módban.

```
Counter dd 0                    ; számláló

IRQ0_Handler Proc

   push eax ds                  ; regiszterek megõrzése

   mov  ax,Data32
   mov  ds,ax
   inc  Counter                 ; a számláló növelése

   mov  ax,Screen
   mov  ds,ax
   inc  byte ptr ds:[0]         ; a bal felsõ sarokban
                                ; lévõ karakter növelése

   mov  al,20h                  ; PIC: End Of Interrupt
   out  20h,al                  ; megszakítás-kezelõ
                                ; végének jelzése

   pop  ds eax
   iretd

IRQ0_Handler Endp
```

És végül a védett módú program, amely 5 másodpercig vár, majd kilép. Fontos, hogy a megszakításokat engedélyezni kell előtte, különben nem generálódik az IRQ 0, vagyis a számláló sem változik, ami pedig végtelen ciklushoz vezetne...

```
ProtSeg  Segment Para Public Use32

Prot_Start:

   sti                          ; megszakítások engedélyezése
Wait_5:
   cmp Counter,91               ; 5 másodperc (18.2*5)
   jb  Wait_5

   cli                          ; megszakítások letiltása újra

   Jump32_16 Code16,Real_Back
                                ; visszaugrás

ProtSeg  Ends
```

1.7. Képernyő kezelése védett módban I.

Jelenleg nem tudunk védett módból valós módú megszakításokat közvetlenül meghívni, így a DOS vagy a BIOS képernyőkezelő eljárásait sem tudjuk használni. Ezért szükséges néhány egyszerű eljárás, amivel védett módban is tudunk a képernyőre írni.

Korábban már definiáltunk egy deszkriptort, ami a szöveges képernyő videomemóriájára mutat, most is ezt használjuk majd fel.

Legelőször szükség van egy eljárásra, amivel karaktereket írhatunk ki adott színnel az aktuális pozícióba. Az alábbi WriteChar eljárás az AL-ben lévő karaktert kiírja a képernyőre, az eljáráshoz szükséges X, Y koordinátákat és az aktuális színt az adatszegmensben tároljuk majd.

```
ProtSeg  Segment

WriteChar Proc

   push eax edx es

   mov  dx,Screen               ; es = képernyõ
   mov  es,dx
```

```
    cmp   al,13                 ; ha 13 -> új sorba lépés
    je    WriteCR

    mov   dl,al                 ; paraméter a dl-be

    mov   eax,80                ; 80 oszlop
    mul   ScreenY               ; szorozva a sorokkal
    add   al,ScreenX            ; plusz az oszlop
    adc   ah,0                  ; túlcsordulás esetére...
    shl   eax,1                 ; *2 a színek miatt

    mov   dh,ScreenColor        ; a szín a dh-ba

    mov   es:[eax],dx           ; karakter kiírása

    inc   ScreenX               ; lépés jobbra
    cmp   ScreenX,80            ; sorvége?
    jae   WriteCR               ; igen, lépjünk a
                                ; következõ sor elejére

WriteCharDone:                  ; végeztünk
    pop   es edx eax
    ret

WriteCR:
    cmp   ScreenY,24            ; a legutolsó sor?
    jae   Scroll                ; igen, görgessük a
                                ; képernyõt
    inc   ScreenY               ; lépés a következõ sor
    mov   ScreenX,0             ; elejére
    jmp   WriteCharDone

Scroll:
    call  ScrollScreen          ; képernyõ görgetése
    mov   ScreenX,0             ; az utolsó sor eleje
    jmp   WriteCharDone

WriteChar Endp


ScreenX     db 0                ; X koordináta
ScreenY     db 0                ; Y koordináta
ScreenColor db 7                ; a szöveg színe


ProtSeg Ends
```

Mint látható, a karakter kiírása után a következő oszlopba lépünk, az utolsó oszlop esetén – vagy ha az AL-ben lévő érték 13 – pedig a következő sor elejére. Ha már az utolsó sor végén vagyunk, akkor egy sorral feljebb

görgetjük a képernyőt. A görgetés megoldható a 2–25. sorok átmásolásával, és az utolsó sor törlésével:

```
ScrollScreen Proc

   push eax ecx esi edi ds es

   mov  ax,Screen            ; ds, es = képernyõ
   mov  ds,ax
   mov  es,ax

   mov  esi,160              ; ds:esi = 2. sor
   mov  edi,0                ; es:edi = 1. sor

   mov  ecx,80*24/2
   rep  movsd                ; 24 sor átmásolása

   mov  ecx,80/2             ; a 25. sor törlése
   mov  eax,07200720h        ; szürke szóközökkel
   rep  stosd

   pop  es ds edi esi ecx eax
   ret

ScrollScreen Endp
```

Hasznos lehet még két kis eljárás, amivel a koordinátákat és az aktuális színt be tudjuk állítani. A GotoXY eljárás a DX-ben megadott sorba (DH) és oszlopba (DL) lép, a SetColor pedig a DL alapján állítja be a színt.

```
GotoXY   Proc

   mov  ScreenY,dh
   mov  ScreenX,dl
   ret

GotoXY   Endp


SetColor Proc

   mov  ScreenColor,dl
   ret

SetColor Endp
```

Mivel a későbbi képernyőkezelő eljárások a WriteChar-t használják, ezért érdemes kipróbálni, hogy jól működik-e. Az alábbi rövid program fel-

tölti a képernyőt az 'A' és a 'Z' közötti karakterekkel, minden sorba más karaktert írva.

```
ProtSeg Segment Para Public Use32

Prot_Start:

    mov  al,'A'                 ; az elsõ karakter
L1:
    mov  cl,80                  ; az oszlopok száma
L2:
    call WriteChar              ; az adott karakter kiírása
    dec  cl
    jnz  L2                     ; a teljes sor kiírása

    inc  al                     ; következõ karakter
    cmp  al,'Z'
    jbe  L1                     ; következõ sor

    Jump32_16 Code16,Real_Back
                                ; visszakapcsolás valós módba

ProtSeg Ends
```

1.8. Képernyő kezelése védett módban II.

A karakter kiírásán kívül alapvető dolog a stringek és a hexadecimális számok kiírása a képernyőre.

Az alábbi eljárás a WriteChar segítségével írja ki a DS:ESI által címzett stringet, amit egy nulla végjel zár le. A legegyszerűbb, ha az új sort csak a 13 jelzi, nem pedig a 13 és 10 (CRLF), mint pl. DOS alatt.

```
WriteStr Proc

     push eax esi

WS1: lodsb                      ; karakter beolvasása
     cmp  al,0                  ; a végére értünk?
     je   WS2                   ; igen.
     call WriteChar             ; a karakter kiírása
     jmp  WS1                   ; jöhet a következõ

WS2: pop  esi eax
     ret

WriteStr Endp
```

A hexadecimális számok kiírása egyszerű, 4 bitenként kell az egyes értékeket számjegyekké (0–9), illetve betűkké (a–f) alakítani. Az alábbi WriteHex eljárás az EDX-ben lévő 32 bites számot jeleníti meg a képernyőn.

```
WriteHex Proc

   push eax edx ecx

   mov  ecx,8                ; 8 számjegy van.

WH_Loop:
   rol  edx,4                ; a következõ számjegy
                             ; az alsó 4 bitbe
   mov  al,dl
   and  al,0fh               ; a felsõ 4 bit törlése
   cmp  al,9                 ; 0 és 9 közötti?
   jle  WH_Dec               ; igen, írjuk ki.
   add  al,'a'-'9'-1         ; kiigazítás betûre

WH_Dec:
   add  al,'0'               ; ASCII karakterré alakítás
   call WriteChar            ; a karakter kiírása
   loop WH_Loop              ; mind a 8 számjegy kiírása

   pop  ecx edx eax
   ret

WriteHex Endp
```

Gyakran előfordul, hogy csak egy 16 bites számot kell kiírni, és ilyenkor zavaró lehet, ha feleslegesen néhány 0 áll a szám elején. Ezért célszerű a WriteHex16 eljárást is bevezetni, amely a DX regiszterben lévő 4 jegyű hexadecimális számot írja ki.

```
WriteHex16 Proc

   push eax edx ecx

   mov  ecx,4                ; 4 számjegy van.

WH_Loop16:
   rol  dx,4                 ; a következõ számjegy
                             ; az alsó 4 bitbe
   mov  al,dl
   and  al,0fh               ; a felsõ 4 bit törlése
   cmp  al,9                 ; 0 és 9 közötti?
   jle  WH_Dec16             ; igen.
```

```
    add  al,'a'-'9'-1          ; kiigazítás betûre

WH_Dec16:
    add  al,'0'                ; ASCII karakterré alakítás
    call WriteChar             ; a karakter kiírása
    loop WH_Loop16             ; mind a 4 számjegy kiírása

    pop  ecx edx eax
    ret

WriteHex16 Endp
```

A képernyőtörlés is könnyen megoldható, mindössze csak szóközökkel fel kell tölteni a teljes képernyőt, és az X, Y koordinátákat a bal felső sarokba állítani.

```
ClearScreen Proc

    push eax ecx edi es

    mov  ax,Screen             ; ES:EDI = képernyõ
    mov  ds,ax
    xor  edi,edi

    mov  eax,07200720h         ; szürke szóközök
    mov  ecx,80*25/2           ; 25 sor
    rep  movsd                 ; törlés

    mov  ScreenX,0             ; (0,0)
    mov  ScreenY,0
    mov  ScreenColor,7         ; szürke szín

    pop  es ds edi ecx eax
    ret

ClearScreen Endp
```

A fenti eljárásokat az alábbi példával ellenőrizhetjük, ami kiír egy 32 illetve 16 bites hexadecimális számot, majd a képernyő közepén elhelyez egy sárga színű szöveget.

```
ProtSeg Segment

Prot_Start:

    call ClearScreen           ; képernyõ törlés

    lea  esi,Ex8_Str
    call WriteStr
```

```
    lea   esi,Hex_Str1
    call  WriteStr

    mov   edx,1234567890        ; 32 bites szám kiírása
    call  WriteHex

    lea   esi,Hex_Str2
    call  WriteStr

    mov   edx,-1
    call  WriteHex16            ; 16 bites szám kiírása

    mov   dl,14
    call  SetColor              ; sárga szín beállítása

    mov   dx,0920h              ; 10. sor közepe
    call  GotoXY

    lea   esi,XY_Str            ; szöveg kiírása
    call  WriteStr

    Jump32_16 Code16,Real_Back
                                ;  visszakapcsolás valós módba


Ex8_Str   db 'Példa 8: képernyõkezelõ funkciók.'
          db 13,0
Hex_Str1  db 'Az 1234567890 decimális szám '
          db 'hexadecimálisan: ',0
Hex_Str2  db 13
          db 'A -1 16 bites hexadecimális értéke: ',0
XY_Str    db 'Ez a 10. sor közepe.',0

ProtSeg Ends
```

1.9. A regiszterek kiírása a képernyőre kivétel esetén

Az eddigi kivételkezelő bármilyen hiba esetén egyszerűen csak megszakította a programot, és visszakapcsolt valós módba. Bár ezzel nagyrészt elkerültük a „lefagyást", a hiba okának kiderítése néha mégis nehéz lehet.

Mivel már tudunk a képernyőre írni, megoldható, hogy kivétel fellépése esetén kiírjuk az összes általános-, és szegmensregiszter aktuális tartalmát. Sőt a hibát okozó utasítás pontos címét is megállapíthatjuk, hisz az EFLAGS, a CS és az EIP tartalma is a verembe kerül.

Először az adott kivétel sorszámát és típusát írjuk majd ki. Ehhez kell néhány string a védett módú szegmensben:

```
Exc_Str0  db '0. kivétel: osztási hiba',0
Exc_Str1  db '1. kivétel: debug',0
Exc_Str2  db '2. kivétel: NMI',0
Exc_Str3  db '3. kivétel: töréspont',0
Exc_Str4  db '4. kivétel: INTO',0
Exc_Str5  db '5. kivétel: BOUND',0
Exc_Str6  db '6. kivétel: érvénytelen utasítás',0
Exc_Str7  db '7. kivétel: a koprocesszor nem '
          db 'elérhetõ',0
Exc_Str8  db '8. kivétel: dupla hiba',0
Exc_Str9  db '9. kivétel: koprocesszor határsértés',0
Exc_Str10 db '10. kivétel: érvénytelen TSS',0
Exc_Str11 db '11. kivétel: verem hiba',0
Exc_Str12 db '12. kivétel: szegmens nincs jelen',0
Exc_Str13 db '13. kivétel: általános védelmi hiba',0
Exc_Str14 db '14. kivétel: laphiba',0
Exc_Str15 db '15. kivétel: fenntartva',0
Exc_Str16 db '16. kivétel: koprocesszor hiba',0
Exc_Str17 db '17. kivétel: illeszkedés ellenõrzés',0
Exc_Str18 db '18. kivétel: processzortípus '
          db 'ellenõrzés',0
```

A kivételek között sok van, amelyek a mostani példákban nagy valószínűséggel nem fordulhatnak elő. Például a koprocesszorral nem foglalkozunk (7-es, 9-es, 16-os kivételek), az INTO, BOUND utasításokat sem használjuk (4–5. kivétel), és most a lapozás sincs engedélyezve (14. illetve 12. kivétel), stb.

A kivételeket kétféleképpen fejezhetjük be: ha nem tudjuk kezelni az adott kivételt, akkor visszakapcsolunk valós módba, és kilépünk (Exc_Abort), ha pedig sikerült a hibát kijavítani, akkor a hibát okozó utasítás újraindítható, tehát folytathatjuk a programot (Exc_OK).

```
Exc_Abort:
   Jump32_16 Code16,Real_Back

Exc_OK:
   iretd
```

A kivételkezelő eljárásokat most úgy alakítjuk át, hogy a verembe rakjuk a kivételhez tartozó string címét, majd meghívjuk az Exc_Show eljárást, ami kiírja ezt a stringet, illetve a regiszterek tartalmát a képernyőre. A töréspont kivétel esetén folytatható a program, mivel ez trap, vagyis a veremben

lévő cím a következő utasításra mutat, és egy IRETD után ez hajtódik majd végre. Ez segíthet a nyomkövetésben is, hiszen ha elhelyezzük a CCh byteot (INT 3) a kódszegmensben, és idekerül a vezérlés, a regiszterek aktuális tartalma kiíródik a képernyőre, majd a következő utasítással folytatódik a végrehajtás. A többi kivételnél pedig megszakítjuk a programot, hisz a felmerülő hibákat még nem tudjuk kezelni.

```
Exc0:
   push offset Pgroup:Exc_Str0
                              ; kivétel leírása a verembe
   call Exc_Show              ; regiszterek kiírása
   jmp  Exc_Abort             ; kilépés a programból
Exc1:
   push offset Pgroup:Exc_Str1
   call Exc_Show
   jmp  Exc_Abort             ; kilépés
Exc2:
   push offset Pgroup:Exc_Str2
   call Exc_Show
   jmp  Exc_Abort             ; kilépés
Exc3:
   push offset Pgroup:Exc_Str3
   call Exc_Show
   jmp  Exc_OK                ; visszatérés a hívóhoz
Exc4:
   push offset Pgroup:Exc_Str4
   call Exc_Show
   jmp  Exc_Abort             ; kilépés
Exc5:
   push offset Pgroup:Exc_Str5
   call Exc_Show
   jmp  Exc_Abort             ; kilépés
Exc6:
   push offset Pgroup:Exc_Str6
   call Exc_Show
   jmp  Exc_Abort             ; kilépés
Exc7:
   push offset Pgroup:Exc_Str7
   call Exc_Show
   jmp  Exc_Abort             ; kilépés
```

A 8–14. és a 17. kivételek esetén egy 32 bites hibakód is kerül a verembe, ezt is érdemes megjeleníteni a képernyőn. Mivel az EBP regisztert ezekben a példákban nem használjuk, ezért ezt a hibakódot berakhatjuk az EBP-be, amit úgyis kiírunk a képernyőre. A hibakód csak 16 bites, emiatt az EBP alsó részét kell figyelembe venni, a felső részében bármilyen érték lehet. Mivel az EBP is gyakran használatos (főleg magasabb szintű nyelveknél),

szükség esetén könnyedén megoldható az is, hogy az EBP felhasználása nélkül írjuk ki a hibakódot a képernyőre.

```
Exc8:
    pop  ebp                    ; hibakód az ebp-be
    push offset Pgroup:Exc_Str8
                                ; kivétel leírása a verembe
    call Exc_Show               ; regiszterek kiírása
    jmp  Exc_Abort              ; kilépés a programból
Exc9:
    pop  ebp                    ; hibakód az ebp-be
    push offset Pgroup:Exc_Str9
    call Exc_Show
    jmp  Exc_Abort              ; kilépés a programból
Exc10:
    pop  ebp                    ; hibakód az ebp-be
    push offset Pgroup:Exc_Str10
    call Exc_Show
    jmp  Exc_Abort              ; kilépés a programból
Exc11:
    pop  ebp                    ; hibakód az ebp-be
    push offset Pgroup:Exc_Str11
    call Exc_Show
    jmp  Exc_Abort              ; kilépés a programból
Exc12:
    pop  ebp                    ; hibakód az ebp-be
    push offset Pgroup:Exc_Str12
    call Exc_Show
    jmp  Exc_Abort              ; kilépés a programból
Exc13:
    pop  ebp                    ; hibakód az ebp-be
    push offset Pgroup:Exc_Str13
    call Exc_Show
    jmp  Exc_OK                 ; folytatás
Exc14:
    pop  ebp                    ; hibakód az ebp-be
    push offset Pgroup:Exc_Str14
    call Exc_Show
    jmp  Exc_Abort              ; kilépés a programból
Exc15:
    pop  ebp                    ; hibakód az ebp-be
    push offset Pgroup:Exc_Str15
    call Exc_Show
    jmp  Exc_Abort              ; kilépés a programból
Exc16:
    push offset Pgroup:Exc_Str16
                                ; itt nincs hibakód
    call Exc_Show
    jmp  Exc_Abort              ; kilépés a programból
Exc17:
    pop  ebp                    ; hibakód az ebp-be
```

```
    push offset Pgroup:Exc_Str17
    call Exc_Show
    jmp  Exc_Abort              ; kilépés a programból
Exc18:
    push offset Pgroup:Exc_Str18
    call Exc_Show
    jmp  Exc_Abort              ; kilépés a programból
```

Most már csak a regiszterek kiírása van hátra. Mivel néhány regisztert (EAX, EDX, ESI, DS) óhatatlanul meg kell változtatni, ezeket a verembe rakjuk, és amikor kiírjuk őket, a verembeli értékeket használjuk fel. A mostani példánál az eljárás hívása és a regiszterek elmentése után a verem a következőképpen néz ki:

kezdeti SS:ESP ⇨	32	a kivétel előtti EFLAGS
	28	a kivételt okozó CS
	24	a kivételt okozó EIP
	20	a string címe
	16	az eljárást hívó EIP
	12	az EAX eredeti tartalma
	8	az EDX eredeti tartalma
	4	az ESI eredeti tartalma
jelenlegi SS:ESP ⇨	0	a DS eredeti tartalma

1.5. ábra A verem tartalma

Az eljárás végén a paraméterként átadott string címét ki kell szedni a veremből, ez például egy ret 4 utasítással megoldható.

A szegmensregiszterekben lévő szelektorok megjelenítésére a WriteHex16 (16 bites hexadecimális szám kiírása) eljárást használjuk fel, a többi regiszter aktuális tartalmát pedig a WriteHex eljárással írjuk ki 32 bites számként.

```
ProtSeg  Segment

Exc_Show Proc

    push eax edx esi ds         ; regiszterek elmentése

    mov  ax,Data32              ; DS = adatszegmens
    mov  ds,ax

    mov  al,13                  ; új sorba lépés
    call WriteChar
```

```
mov  esi,ss:[esp+20]      ; az adott kivétel kiírása
mov  ScreenColor,15       ; fehér színnel
call WriteStr
mov  ScreenColor,7

mov  al,13
call WriteChar            ; 1 sor kihagyás
call WriteChar

lea  esi,Str_eax          ; EAX
call WriteStr
mov  edx,ss:[esp+12]
call WriteHex

lea  esi,Str_ds           ; DS
call WriteStr
mov  edx,ss:[esp]
call WriteHex16

lea  esi,Str_esi          ; ESI
call WriteStr
mov  edx,ss:[esp+4]
call WriteHex

lea  esi,Str_fs           ; FS
call WriteStr
mov  dx,fs
call WriteHex16

mov  al,13                ; új sor
call WriteChar

lea  esi,Str_ebx          ; EBX
call WriteStr
mov  edx,ebx
call WriteHex

lea  esi,Str_es           ; ES
call WriteStr
mov  dx,es
call WriteHex16

lea  esi,Str_edi          ; EDI
call WriteStr
mov  edx,edi
call WriteHex

lea  esi,Str_gs           ; GS
call WriteStr
mov  dx,gs
call WriteHex16
```

```
    mov  al,13                  ; új sor
    call WriteChar

    lea  esi,Str_ecx            ; ECX
    call WriteStr
    mov  edx,ecx
    call WriteHex

    lea  esi,Str_ss             ; SS
    call WriteStr
    mov  dx,ss
    call WriteHex16

    lea  esi,Str_esp            ; ESP
    call WriteStr
    mov  edx,esp
    call WriteHex

    lea  esi,Str_ebp            ; EBP
    call WriteStr
    mov  edx,ebp
    call WriteHex

    mov  al,13                  ; új sor
    call WriteChar

    lea  esi,Str_edx            ; EDX
    call WriteStr
    mov  edx,ss:[esp+8]
    call WriteHex

    lea  esi,Str_cs             ; CS
    call WriteStr
    mov  dx,ss:[esp+28]
    call WriteHex16

    lea  esi,Str_eip            ; EIP
    call WriteStr
    mov  edx,ss:[esp+24]
    call WriteHex

    mov  al,13                  ; új sor
    call WriteChar

    pop  ds esi edx eax

    ret  4                      ; paraméter kiszedése a veremből.

Exc_Show Endp
```

```
Str_eax db 'EAX = ',0
Str_ebx db 'EBX = ',0
Str_ecx db 'ECX = ',0
Str_edx db 'EDX = ',0

Str_esi db '       ESI = ',0
Str_edi db '       EDI = ',0
Str_esp db '       ESP = ',0
Str_eip db '       EIP = ',0

Str_ds  db '       DS = ',0
Str_es  db '       ES = ',0
Str_ss  db '       SS = ',0
Str_cs  db '       CS = ',0

Str_fs  db '       FS = ',0
Str_gs  db '       GS = ',0
Str_ebp db '       EBP= ',0

ProtSeg Ends
```

A fentieket kipróbálhatjuk, ha például osztási hibát, vagy érvénytelen utasítást generálunk. Hogy jól látható legyen az eredmény, a regisztereket előtte feltöltjük bizonyos értékekkel.

```
ProtSeg Segment Para Public Use32

Prot_Start:

   lea  esi,Ex9_Str          ; szöveg kiírása
   call WriteStr

   mov  eax,11111111h        ; egyszerû számok a
   mov  ebx,22222222h        ; regiszterekbe
   mov  ecx,33333333h
   mov  edx,44444444h
   mov  esi,55555555h
   mov  edi,66666666h
   mov  ebp,77777777h

   xor  bl,bl                ; osztás nullával
   div  bl

   Jump32_16 Code16,Real_Back    ; visszaugrás?

Ex9_Str db 'Példa 9: kivételkezelés',13,0

ProtSeg Ends
```

A 0-val való osztás miatt a 0. kivétel kezelője hívódik meg, amely kiírja a regisztereket a képernyőre, majd visszakapcsol valós módba. Az általános regiszterekben az általunk megadott értékek vannak (illetve BL = 0), a szegmensregiszterek a kódszegmens szelektort tartalmazzak (Code32, 0018h), a verembe elmentett CS:EIP pedig a hibát okozó utasításra mutat.

```
EAX=11111111   DS=0018   ESI=55555555   FS=18
EBX=22222200   ES=0018   EDI=66666666   GS=18
ECX=33333333   SS=0028   ESP=00000d7c   EBP=77777777
EDX=44444444   CS=0008   EIP=000002cf
```

A nullával való osztás mellett a 0. kivételt könnyedén létrehozhatjuk osztási túlcsordulással (pl. 500/5), vagy akár a 0. megszakítás meghívásával (INT 0). A 6. kivételt is kipróbálhatjuk például a LOCK NOP utasítással.

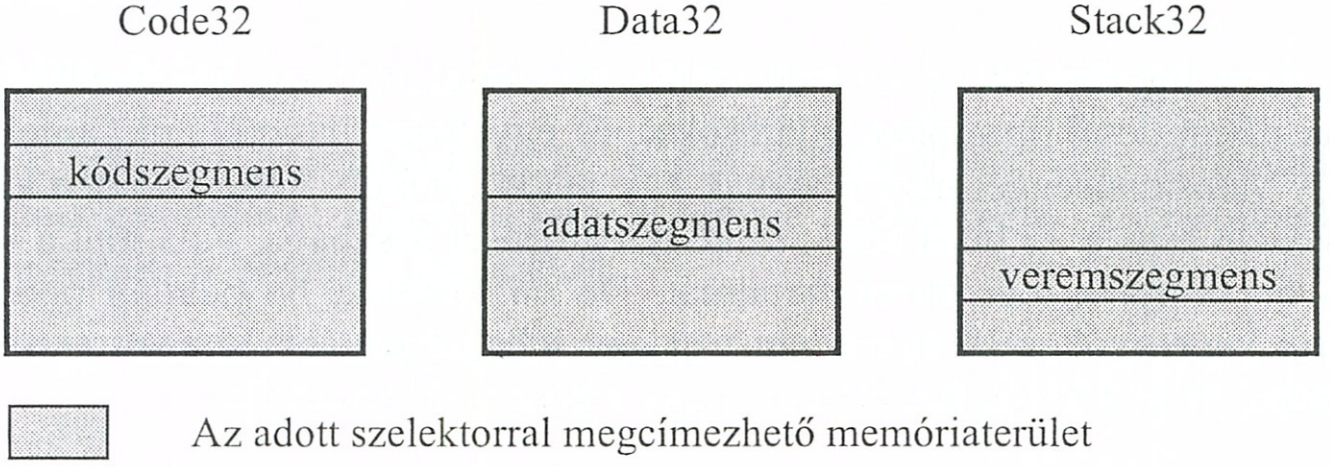
Arra figyelni kell a kivételkezelők megírásakor, hogy ha lehet, akkor csak olyan deszkriptorokat, eljárásokat, és egyebeket használjunk, amelyek nagy valószínűséggel nem okoznak egy újabb kivételt. Most minden kivételkezelő csak a 32 bites adat-, és kódszegmenset használja, valamint a képernyőre kiírja a regiszterek tartalmát. De mi történik, ha például pont az egyik képernyőkezelő eljárásban van egy hiba? Ez újabb kivételt eredményezne, de ez a kivételkezelő ugyanúgy a képernyőt használja...

Az enyhébb kivételek esetén mindaddig meghívódik az újabb kivétel, amíg a hiba ki nem lett javítva, vagy amíg a verem be nem telik, ez pedig veremhiba kivételt okoz. A komolyabb kivételek kezelésére viszont már nem elég a mostani módszer, ezek már dupla, esetleg tripla hibát okoznának, ha nem megfelelő módon kezeljük őket. A következő részben a taszkokról is szó lesz, amelyek segítségével akár bizonyos kivételeket (pl. veremhiba) már taszkkapukon keresztül is kezelni lehet, és ezáltal tovább növelhető a védett módú programok biztonsága.

2. A taszkok és a védelem

2.1. A szegmensszintű védelem

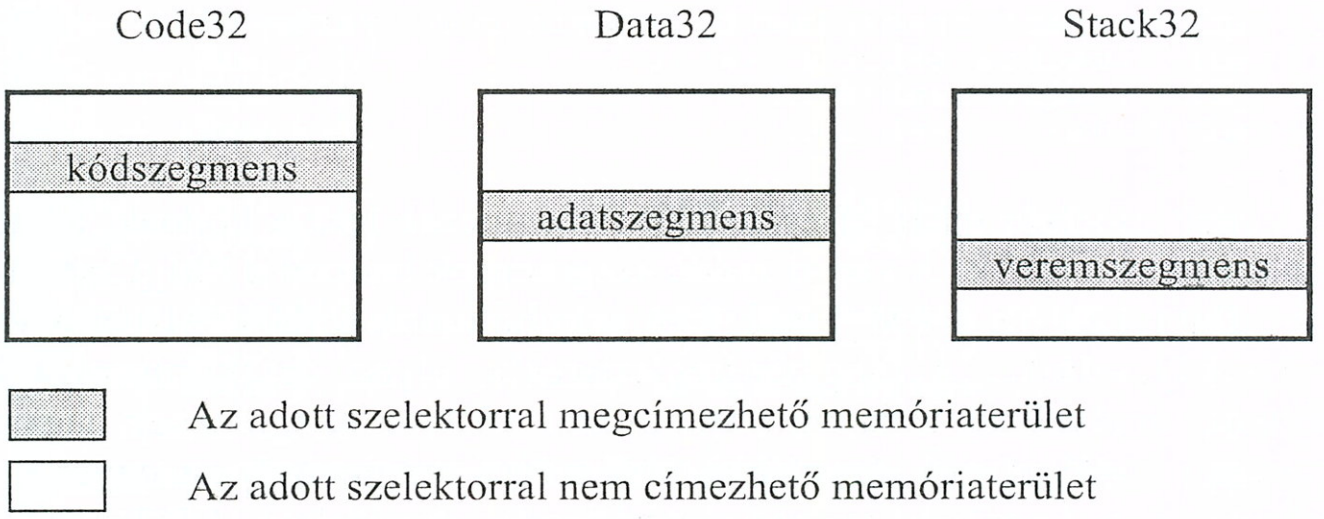
Az eddigi példákban a különböző 32 bites szegmensdeszkriptorok 64 KB nagyságúak voltak, és a báziscímük is megegyezett, ami ugyan megkönnyítette a deszkriptortáblázat beállítását, viszont így a program sokat vesztett a biztonságából. Például az adatszegmens segítségével a kódszegmenset, illetve a veremhatár túllépésével a kód-, vagy az adatszegmenset felül lehetett volna írni.



2.1. ábra Az eddigi deszkriptorok

Ezért mostantól minden deszkriptor nagysága éppen akkora lesz, mint a hozzá tartozó szegmensé, valamint az adat-, a kód-, és a veremszegmensek külön memóriapozíción lesznek, azaz más lesz a báziscímük is. Így elkerülhető, hogy egy hibás program átírjon olyan memóriaterületet, amit nem lenne szabad, illetve ami nem hozzá tartozik. Minden ilyen kísérlet általános védelmi hibát (13. kivétel) fog okozni.

A deszkriptorok definiálásánál tehát tudni kell az egyes szegmensek pontos hosszát, amely megállapítható például a szegmensek legvégén elhelyezett címkék segítségével. Ahhoz, hogy egyértelműen meghatározható legyen a szegmensek hossza, a main.asm file elején és végén szerepelnek majd a szegmensek definíciói, illetve a hosszukat jelölő szimbólumok. Ez a file tartalmazza majd az include file-okat is, ahol a GDT, az IDT, a képernyőkezelés, stb. található.



2.2. ábra A deszkriptorok mostantól kezdve

main.asm

```
RM_Seg Segment Use16          ; valós módú szegmens
RM_Seg_Start Label Byte
RM_Seg Ends

PM_Code Segment Use32         ; védett módú kódszegmens
PM_Code_Start Label Byte
PM_Code Ends

PM_Data Segment Use32         ; védett módú adatszegmens
PM_Data_Start Label Byte
PM_Data Ends

; itt lesznek az include file-ok,
; átkapcsolás védett módba, stb.

RM_Seg Segment
RM_Seg_Len   Equ $-RM_Seg_Start
RM_Seg Ends

PM_Code Segment
PM_Code_Size Equ $-PM_Code_Start
PM_Code Ends

PM_Data Segment
PM_Data_Size Equ $-PM_Data_Start
PM_Data Ends


RM_Stack Segment Stack Use16 ; valós módú verem (1K)
RM_Stack_Size Equ 400h
db RM_Stack_Size dup (?)
RM_Stack Ends
```

```
PM_Stack Segment Stack Use32 ; védett módú verem (1K)
PM_Stack_Size Equ 400h
db PM_Stack_Size dup (?)
PM_Stack Ends
```

Védett módban tehát teljesen külön szegmense lesz a kódnak, az adatoknak, és a veremnek. A védett módú szegmensek hosszát a PM_Code_Len, a PM_Data_Len, és a PM_Stack_Len szimbólumok jelölik. Mivel a valós módba kapcsoláskor a szegmensregiszterekbe olyan deszkriptorokat kell betölteni, amelyeknek a határa 64K, ezért a 16 bites szegmensek nagysága továbbra is 64K marad. Az alábbi táblázatban szerepelnek az egyes szegmensek, és a hozzájuk tartozó szelektorok, amelyek a GDT-ben lévő deszkriptorokra mutatnak.

Szelektor	Szegmens neve	Leírás
Code32	PM_Code	védett módú kódszegmens
Data32	PM_Data	védett módú adatszegmens
Stack32	PM_Stack	védett módú veremszegmens
Code16	RM_Seg	valós módú kódszegmens
Data16	RM_Seg	valós módú adatszegmens
—	RM_Stack	valós módú veremszegmens

2.3. ábra Szegmensek és szelektorok

A veremszegmenseket célszerű legutoljára definiálni, mivel így nem foglalnak helyet az EXE file-ban. Ha nem ezek az utolsó szegmensek, akkor a db (?) használata ellenére is felesleges 0-kat tartalmazna a file.

2.1.1. Szimbólum-definíciók

Legjobb lenne a deszkriptortáblázatokat úgy átalakítani, hogy a közvetlen hexadecimális számok helyett szimbólumokat használatával építhessük fel az egyes deszkriptorokat. Ezeket a szimbólumokat a defs.inc file-ba helyezzük majd el. Nézzük először a deszkriptorokban lévő P, DPL és S, valamint a G és D/B biteket:

defs.inc

```
Present    Equ 80h          ; a memóriában jelenlévõ szegmens
NotPresent Equ 0            ; nem jelenlévõ szegmens
```

```
DPL0        Equ 0           ; 0. szintû deszkriptorok
DPL1        Equ 20h         ; 1. szintû deszkriptorok
DPL2        Equ 40h         ; 2. szintû deszkriptorok
DPL3        Equ 60h         ; 3. szintû deszkriptorok

Seg_Desc    Equ 10h         ; szegmensdeszkriptor
Sys_Desc    Equ 0           ; rendszerdeszkriptor

Seg_Big     Equ 80h         ; nagyfelbontású szegmens
Seg_32      Equ 40h         ; 32 bites szegmens
Seg_16      Equ 0           ; 16 bites szegmens
```

A szegmensdeszkriptorok (S = 1) esetén a típus mezőnél több variáció is lehetséges, attól függően, hogy a deszkriptor egy kód-, vagy egy adatszegmenset jelöl, illetve, hogy a szegmens írható és olvasható-e. Továbbá a kódszegmensek lehetnek illeszkedőek is, az adatszegmensek pedig lefelé bővülőek (veremszegmensek esetén).

```
Code_EO     Equ 8           ; csak végrehajtható kódszegmens
                            ; (Execute Only)
Code_ER     Equ 10          ; végrehajtható, olvasható kódsz.
                            ; (Execute, Read)
Code_EOC    Equ 12          ; végrehajtható, illeszkedõ kódsz.
                            ; (Execute Only, Conforming)
Code_ERC    Equ 14          ; végrehajtható, olvasható,
                            ; illeszkedõ kódszegmens
                            ; (Execute, Read, Conforming)

Data_RO     Equ 0           ; csak olvasható adatszegmens
                            ; (Read Only)
Data_RW     Equ 2           ; olvasható és írható adatsz.
                            ; (Read, Write)
Data_ROED   Equ 4           ; csak olvasható, lefelé bõvülõ
                            ; (Read Only, Expand Down)
Data_RWED   Equ 6           ; írható, olvasható, lefelé bõvülõ
                            ; (Read, Write, Expand Down)
```

A rendszerdeszkriptorok közül csak az alábbiak fontosak számunkra:

```
Sys_LDT         Equ 2       ; lokális deszkriptortáblázat
Sys_TSS         Equ 9       ; taszk-állapotszegmens
Sys_TaskGate    Equ 5       ; taszkkapu
Sys_CallGate    Equ 12      ; call-kapu
Sys_IntGate     Equ 14      ; megszakításkapu
Sys_TrapGate    Equ 15      ; trapkapu
```

Egy deszkriptort az egyes mezőknek megfelelő szimbólumok összeadásával kaphatunk meg. Például egy 32 bites, 3. privilégiumszintű, jelenlévő,

nagy felbontású, írható és olvasható adatszegmens típusa a Seg_32+DPL3+ +Present+Seg_Big+Data_RW szimbólumok összege lenne, vagyis C0E2h. Hasonló módon egy 1. szintű taszk kapu típusa A5h, amely a Present+ +DPL1+Sys_Desc+Sys_TaskGate összegéből adódik.

Néhány gyakran használt deszkriptor típusát akár előre is definiálhatjuk, hogy ne kelljen mindig az összes bitre hivatkozni, és hogy minél rövidebb legyen a felsorolás. Jelenleg csak a 0. szintű végrehajtható/olvasható kódszegmens, és az írható/olvasható adatszegmens típusaira, illetve a 0. szintű megszakításkapura van szükségünk.

```
Code_Desc     Equ     Present+DPL0+Seg_Desc+Code_ER
Data_Desc     Equ     Present+DPL0+Seg_Desc+Data_RW
Int_GateDesc  Equ     Present+DPL0+Sys_Desc+Sys_IntGate
```

2.1.2. Az új GDT

A szimbólumok bevezetésével a deszkriptortáblázatok könnyebben áttekinthetőek és módosíthatóak lesznek. Mostantól kezdve a GDT is egy külön szegmensbe kerül, a GDT_Seg-be, így elkerülhető, hogy egy hiba miatt véletlenül (esetleg szándékosan) felülíródjon. Korábban ez könnyen előfordulhatott volna. A GDT-ben lévő deszkriptorok definiálásához továbbra is az eddig használt Descriptor makrót használjuk fel. Az eddigi deszkriptorokat kiegészíthetjük egy 0. báziscímű, 4 GB méretű 32 bites adatszegmenssel (Flat32), amellyel szükség esetén a teljes lineáris memória elérhető.

gdt.inc

```
GDT_Seg Segment Para Public

GDT  Label Qword

Descriptor NullSel,0,0,0
Descriptor Code32, 0,PM_Code_Size,    Code_Desc+Seg_32
Descriptor Data32, 0,PM_Data_Size,    Data_Desc+Seg_32
Descriptor Stack32,0,PM_Stack_Size,   Data_Desc+Seg_32
Descriptor Code16, 0,0ffffh,          Code_Desc+Seg_16
Descriptor Data16, 0,0ffffh,          Data_Desc+Seg_16
Descriptor Screen, 0b8000h,4000,      Data_Desc+Seg_32
Descriptor Flat32, 0,0fffffh, Data_Desc+Seg_32+Seg_Big

GDT_Len     Equ $-GDT          ; a GDT hossza

GDT_Value   Label Pword        ; az LGDT utasítás paramétere
GDT_Limit   dw GDT_Len         ; a GDT hossza
```

```
GDT_Base    dd GDT              ; a GDT báziscíme

GDT_Seg     Ends
```

Természetesen most is szükség van arra, hogy a program aktuális memóriacímétől függően a báziscímeket lineáris címekké alakítsuk.

Mivel a különböző típusú szegmenseket teljesen különválasztottuk, ezért most a szegmensdeszkriptorokban egymástól eltérő báziscímek szerepelnek, célszerű tehát itt egy újabb makrót használni, amely a valós módú szegmenseket védett módú báziscímmé alakítja (szorzás 16-al), és ez alapján megváltoztatja a deszkriptorokat. A Seg2Base (Segment To Base) makró paramétere egy valós módú szegmens, az adott deszkriptor címét pedig az ES:DI tartalmazza. Egy deszkriptor báziscímének átalakítása után az ES:DI a táblázat következő bejegyzésére mutat.

```
Seg2Base Macro RealSeg
   xor  eax,eax
   mov  ax,RealSeg          ; szegmens->32 bites báziscím
   shl  eax,4
   add  di,2
   stosw                    ; a Bázis 0-15 bitek tárolása
   shr  eax,16              ; a felsõ 16 bit az alsó 16-ba
   stosb                    ; a Bázis 16-23 bitek tárolása
   add  di,2
   mov  al,ah
   stosb                    ; a Bázis 24-31 bitek tárolása
Endm
```

A GDT-vel kapcsolatban már csak egy valós módú eljárásra van szükség, amely beállítja a deszkriptortáblázatot, és inicializálja a GDTR-t. Az eljárás során az ES szegmensregiszter a GDT saját szegmensére mutat, ezt érdemes tudatni az assemblerrel is (Assume).

```
RM_Seg Segment
      Assume cs:RM_Seg, ds:RM_Seg, es:GDT_Seg

InitGDT Proc

   mov  ax,GDT_Seg          ; ES:DI = a GDT
   mov  es,ax
   lea  edx,GDT_Seg:GDT

   add  dx,8                ; a nulla deszkriptor átugrása
```

```
    Seg2Base PM_Code          ; Code32
    Seg2Base PM_Data          ; Data32
    Seg2Base PM_Stack         ; Stack32
    Seg2Base RM_Seg           ; Code16
    Seg2Base RM_Seg           ; Data16

    mov  eax,GDT_Seg
    shl  eax,4
    add  GDT_Base,eax         ; a GDT_Base fizikai címmé
                              ; alakítása

    lgdt GDT_Value            ; a GDTR beállítása

    ret

InitGDT Endp
```

2.1.3. Az új IDT

A GDT-hez hasonlóan a megszakítások deszkriptorai is egy külön szegmensben lesznek, az IDT_Seg-ben. A megszakításkapuk létrehozására használhatjuk a korábbi Int_Gate makrót, most egyelőre csak a kivételekre van szükségünk, a megszakítások a program során le lesznek tiltva.

idt.inc

```
IDT_Seg Segment Para Public

IDT Label QWord

    ; A kivételek kapui

    Int_Gate 0,Code32,PM_Code:Exc_0
    Int_Gate 1,Code32,PM_Code:Exc_1
    Int_Gate 2,Code32,PM_Code:Exc_2
    Int_Gate 3,Code32,PM_Code:Exc_3
    Int_Gate 4,Code32,PM_Code:Exc_4
    Int_Gate 5,Code32,PM_Code:Exc_5
    Int_Gate 6,Code32,PM_Code:Exc_6
    Int_Gate 7,Code32,PM_Code:Exc_7
    Int_Gate 8,Code32,PM_Code:Exc_8
    Int_Gate 9,Code32,PM_Code:Exc_9
    Int_Gate 0ah,Code32,PM_Code:Exc_10
    Int_Gate 0bh,Code32,PM_Code:Exc_11
    Int_Gate 0ch,Code32,PM_Code:Exc_12
    Int_Gate 0dh,Code32,PM_Code:Exc_13
    Int_Gate 0eh,Code32,PM_Code:Exc_14
    Int_Gate 0fh,Code32,PM_Code:Exc_15
    Int_Gate 10h,Code32,PM_Code:Exc_16
    Int_Gate 11h,Code32,PM_Code:Exc_17
    Int_Gate 12h,Code32,PM_Code:Exc_18
```

```
IDT_Len  Equ                    $-IDT

IDT_Value Label  Pword                  ; az IDTR paramétere
IDT_Limit dw     IDT_Len                ; 16 bites határ
IDT_Base  dd     IDT                    ; 32 bites báziscím

Real_IDT_Value Label Pword    ; az IDTR paramétere
Real_IDT_Limit dw 3ffh                  ; 16 bites határ
Real_IDT_Base  dd 0                     ; 32 bites báziscím

IDT_Seg Ends
```

Az IDT beállítása hasonlít a korábbi példákhoz, viszont most figyelembe kell venni, hogy az IDT_Base és az IDT_Value az IDT_Seg-ben van, illetve, hogy a kivételkezelők a PM_Code szegmensben vannak. Ezt szerencsére az assume direktíva használata megkönnyíti.

```
RM_Seg Segment
      Assume cs:RM_Seg, ds: RM_Seg, es: IDT_Seg

InitIDT Proc

   xor   eax,eax
   mov   ax,IDT_Seg
   mov   es,ax                  ; ES=IDT_Seg

   shl   eax,4
   add   IDT_Base,eax           ; az IDT_Base átalakítása
                                ; fizikai címmé

   lidt IDT_Value               ; IDTR beállítása

   ret

InitIDT Endp

RestoreIDT Proc

   mov   ax,IDT_Seg
   mov   es,ax
   lidt Real_IDT_Value          ; valós módú IDT

   ret

RestoreIDT Endp

RM_Seg Ends
```

Eddig egy védett módú szegmenset használtunk (ProtSeg), amelyben az adat és a kód vegyesen volt elhelyezve. Ezeket most a nagyobb biztonság érdekében szét kell választani, az eddigi eljárások a PM_Code szegmensbe kerülnek, az adatok pedig a PM_Data szegmensbe. Például a képernyőkezelés a következőképpen alakul:

screen.inc

```
PM_Data Segment

ScreenX         db 0            ; X koordináta
ScreenY         db 0            ; Y koordináta
ScreenColor     db 7            ; a szöveg színe

PM_Data Ends

PM_Code Segment
        Assume cs: PM_Code, ds: PM_Data

; WriteChar, WriteString, és a többi eljárás...

PM_Code Ends
```

A kivételeket jelenleg két csoportba oszthatjuk: vannak, amelyek egy hibakódot helyeznek el a veremben a kivétel generálódásakor, és vannak, amelyek nem. Célszerű tehát a kivételkezelőket leegyszerűsíteni erre a két esetre. Az alábbi Exc_Handler makró első paraméterének az adott kivétel sorszámát kell megadni. Ezután egy szám következik, amely jelzi, hogy van-e hibakód (1), vagy nincs (0). A hibakódot a kivétel leírásával együtt a verembe rakjuk az Exc_Show meghívása előtt, és a regiszterek kiírásakor ezt hibakódot is kiírjuk a képernyőre. Ha nincs hibakód, akkor egy 0-t helyezünk a verembe. Az Exc_Show majd csak akkor fogja megjeleníti a hibakódot, ha az nem 0 (pl. a 13. kivétel hibakódja is gyakran 0).

```
Exc_Handler Macro Num,Code
Exc_&Num:
    if Code eq 0
    pop  0                      ; nincs hibakód
    else
    push dword ptr ss:[esp+16]
                                ; a hibakód újra a verembe
    endif

    push offset PM_Data:Exc_Str&Num
    call Exc_Show               ; regiszterek kiírása
    jmp  Exc_Abort              ; kilépés a programból
Endm
```

A védelem miatt a kivételkezelő eljárásoknál is külön kell választani az adatszegmenset és a kódszegmenset. A legjobb az lenne, ha a kivételkezelőknek saját kód-, és adatszegmensük lenne, de most még elég, ha a Code32, illetve a Data32 szelektor tartozik hozzájuk. A töréspont kivétel esetén az eddigiekhez hasonlóan a regiszterek kiírása után nem lépünk ki a programból, hanem a vezérlés a kivételt okozó utasítás után fog folytatódni.

idt.inc

```
PM_Code Segment
        Assume cs:PM_Code, ds:PM_Data, es:Nothing

; a kivételkezelõk (Exc_0 - Exc_18)

    Exc_Handler 0,0
    Exc_Handler 1,0
    Exc_Handler 2,0
    Exc_Handler 4,0
    Exc_Handler 5,0
    Exc_Handler 6,0
    Exc_Handler 7,0
    Exc_Handler 8,1
    Exc_Handler 9,1
    Exc_Handler 10,1
    Exc_Handler 11,1
    Exc_Handler 12,1
    Exc_Handler 13,1
    Exc_Handler 14,1
    Exc_Handler 15,0
    Exc_Handler 16,0
    Exc_Handler 17,1
    Exc_Handler 18,0

Exc3:
    push 0                                  ; nincs hibakód
    push offset PM_Data:Exc_Str3            ; kivétel string
    call Exc_Show                           ; regiszterek kiírása
    iretd                                   ; visszatérés


; kezelhetetlen kivétel, a program megszakítása

Exc_Abort:
    Jump32_16 Code16,Exit_RM
                        ; visszaugrás a 16 bites kódszegmensre

PM_Code Ends

PM_Data Segment
```

```
ExcStr0 db '0. kivétel: osztás nullával.',0
    .
    .
    .

PM_Data Ends
```

Az Exc_Show eljárást egy kicsit át kell alakítani, hogy kiírja a veremben elhelyezett hibakód másolatát, de csak akkor, ha nem 0. A veremben lévő értékek az alábbiak szerint módosulnak:

kezdeti SS:ESP ⇨

36	a kivétel előtti EFLAGS
32	a kivételt okozó CS
28	a kivételt okozó EIP
24	a hibakód másolata, vagy 0
20	a string címe
16	az eljárást hívó EIP
12	az EAX eredeti tartalma
8	az EDX eredeti tartalma
4	az ESI eredeti tartalma
0	a DS eredeti tartalma

jelenlegi SS:ESP ⇨

2.4. ábra A verem tartalma

Mivel a hibakódot a vermen keresztül adjuk át az Exc_Show eljárásnak, ezért az EBP is változatlan marad, és így már a kivételkezelőkből észrevétlenül visszaadhatjuk a vezérlést, ha a hibát kijavítottuk. A hibakód miatt az eljárás végén 2 duplaszót kell majd leemelni a veremből.

```
Exc_Show Proc

    push eax edx esi ds         ; regiszterek elmentése

    mov  ax,Data32              ; DS = adatszegmens
    mov  ds,ax

    mov  al,13                  ; új sorba lépés
    call WriteChar

    mov  esi,ss:[esp+20]        ; az adott kivétel kiírása
    mov  ScreenColor,15         ; fehér színnel
    call WriteStr

    mov  edx,ss:[esp+24]        ; hibakód az EDX-be
    or   edx,edx
```

```
    je    Show_NoErrorCode      ; ha nulla, ne írjuk ki

    lea   esi,Str_Code
    call  WriteStr              ; üzenet kiírása
    call  WriteHex              ; hibakód kiírása

Show_NoErrorCode:
    mov   ScreenColor,7         ; újra szürke szín

    mov   al,13
    call  WriteChar             ; 1 sor kihagyás
    call  WriteChar

    lea   esi,Str_eax           ; EAX
    call  WriteStr
    mov   edx,ss:[esp+12]
    call  WriteHex

    .
    .
    .

    lea   esi,Str_cs            ; CS
    call  WriteStr
    mov   dx,ss:[esp+36]
    call  WriteHex16

    lea   esi,Str_eip           ; EIP
    call  WriteStr
    mov   edx,ss:[esp+32]
    call  WriteHex

    mov   al,13                 ; új sor
    call  WriteChar

    pop   ds esi edx eax

    ret   8                ; paraméterek kiszedése a veremből.

Exc_Show Endp
```

Az eddigiekhez hasonlóan a védett módba kapcsolás előtt most is ellenőrizzük a 386-os processzort, és hogy valós üzemmódban vagyunk-e, beállítjuk a deszkriptortáblázatokat és a megszakításokat, majd a 32 bites kódszegmensre ugrunk (PM_Code_Start). A valós módba való visszatéréskor (Exit_RM) az összes szegmensregiszterbe 64 KB méretű szegmensdeszkriptort helyezünk el, és a megszakításokat is visszaállítjuk a valós módnak megfelelően (báziscím 0, határ 3ffh). Kilépés előtt a BIOS segítségével a kurzort a 20. sorba mozgatjuk, hogy ne íródjon felül, amit védett

módban írtunk a képernyőre. (Vagy akár a képernyőkezelést is módosíthatnánk, hogy a Flat32 használatával a védett módú pozícióval együtt a BIOS adatterületének frissítése is megtörténjen.)

main.inc

```
RM_Seg Segment Para Public Use16
RM_Seg_Start Label Byte
      Assume cs:RM_Seg, ds:RM_Seg

RM_Start:

   cli                          ; megszakítások letiltása

   mov  ax,RM_Seg               ; adatszegmens beállítása
   mov  ds,ax

   call Check386                ; 386-os ellenörzése
   call CheckV86                ; V86 mód ellenörzése

   call InitGDT                 ; GDT beállítása
   call InitIDT                 ; IDT beállítása
   call InitPIC                 ; PIC beállítása

   mov  eax,CR0                 ; átkapcsolás védett módba
   or   al,1
   mov  CR0,eax

   Jump16_32 Code32,PM_Code_Start
                                ; ugrás a 32 bites kódszegmensre

Exit_RM:

   cli                          ; megszakítások letiltása

   mov  ax,Data16               ; a szegmensregiszterek
   mov  ds,ax                   ; feltöltése 64 KB határú
   mov  es,ax                   ; adatszegmens szelektorával
   mov  fs,ax
   mov  gs,ax
   mov  ss,ax

   mov  eax,CR0                 ; visszakapcsolás valós módba
   and  al,not 1
   mov  CR0,eax

   Jump16_16 RM_Seg,Exit_RM2
                     ; valós módú szegmens betöltése a CS-be
```

```
Exit_RM2:

    mov   ax,RM_Seg             ; valós módú érték a
    mov   ds,ax                 ; szegmensregiszterekbe
    mov   es,ax
    mov   fs,ax
    mov   gs,ax

    mov   ax,RM_Stack           ; valós módú verem
    mov   ss,ax                 ; visszaállítása
    mov   sp,RM_Stack_Size

    call  RestoreIDT            ; valós módú megszakítások
    call  RestorePIC            ; az IRQ-k visszaállítása

    mov   ah,2                  ; kurzor-pozíció beállítása
    mov   bh,0
    mov   dx,1400h              ; a 20. sor elejére
    int   10h

    mov   ah,4ch
    int   21h

RM_Seg Ends
```

A 32 bites kódszegmensre való ugrás után az egyes szegmensregisztereket feltöltjük a megfelelő szelektorokkal, beállítjuk a védett módú vermet, majd egy piros-fehér színű fejléc kiírása után töröljük az összes általános regisztert, és meghívjuk a 32 bites főprogramot (PM_Main). A főprogramból való visszatérés után (ha nem történt kivétel) visszakapcsolunk valós módba, és kilépünk a programból.

pm.inc

```
PM_Code Segment Para Public Use32
     Assume cs: PM_Code, ds:PM_Data

    mov   ax,Data32             ; DS, ES = adatszegmens
    mov   ds,ax
    mov   es,ax
    mov   ax,Flat32             ; FS = teljes memória
    mov   fs,ax
    mov   ax,Screen             ; GS = képernyõ-memória
    mov   gs,ax

    mov   ax,Stack32            ; SS:ESP = védett módú verem
    mov   ss,ax
    mov   esp,PM_Stack_Size
    mov   ebp,esp
```

```
    call ClearScreen            ; képernyõtörlés

    mov  dl,4*16+15             ; piros-fehér szín
    call SetColor
    lea  esi,Header             ; fejléc kiírása
    call WriteStr
    mov  dl,7                   ; újra szürke szín
    call SetColor

    xor  eax,eax                ; az általános regiszterek
    xor  ebx,ebx                ; törlése
    xor  ecx,ecx
    xor  edx,edx
    xor  esi,esi
    xor  edi,edi

    call PM_Main                ; a fõprogram hívása

    Jump32_16 Code16,Exit_RM ; kilépés


PM_Code Ends

PM_Data Segment Para Public Use32
PM_Data_Start Label Byte

Header  db 23 dup (32)
        db 'Védett módú programozás gyakorlata'
        db 23 dup (32)
        db 13,13,0

PM_Data Ends
```

A most kialakított biztonságosabb védett módú program kipróbálására a töréspont és az általános védelmi hiba kivételeket használjuk fel. Az INT 3 hatására a regiszterek kiíródnak a képernyőre, és a program végrehajtása folytatódik tovább, ez ideális a nyomkövetésre és a hibakeresésre.

Ezután a DS-be 100h-t töltünk be, hogy ellenőrizzük a hibakódok kezelését. A GDT-ben jelenleg csak 8 deszkriptor van (a GDTR-ben lévő határ 64 byte), és a 100h szelektor a 32. deszkriptorra utal, ezért az utasítás végrehajtásakor 13. kivétel keletkezik. Mivel a hibát egy szegmensregiszter okozta, ezért a hibakód az az érték lesz, amit megpróbáltunk betölteni a DS-be.

A szegmensszintű védelmet is könnyen ellenőrizhetjük, ha megpróbálunk az adatszegmens határán kívüli adatot elérni, vagy ha a végrehajtás során a kódszegmens határát túllépjük.

```
PM_Code Segment Para
     Assume cs: PM_Code, ds:PM_Data, es:PM_Data
PM_Main:

   lea  esi,Ex1_Str
   call WriteStr

   mov  eax,11111111h        ; regiszterek feltöltése
   mov  ebx,22222222h
   mov  ecx,33333333h
   mov  edx,44444444h
   mov  esi,55555555h
   mov  edi,66666666h

   int 3                     ; regiszterek kiírása

   add  ebx,eax              ; kis változtatások
   sub  ecx,eax
   and  edx,eax
   or   esi,eax
   xor  edi,eax

   push 100h
   pop  ds                   ; DS=100h, 13. kivétel


   mov  eax,EndOfData        ; most már ezek is kivételt
   jmp  EndOfCode            ; generálnak

   ret

EndOfCode:
   nop                       ; ez a kódszegmens vége

PM_Code Ends



PM_Data Segment

Ex1_Str   db '1. Példa: a 3. és 13. kivétel '
          db 'tesztelése.',13,0

EndOfData Label Dword        ; itt van az adatszegmens vége

PM_Data Ends
```

A pop ds utasítás hatására a kivételkezelő a következőket írja ki a képernyőre:

```
13. kivétel: általános védelmi hiba, hibakód: 00000100

EAX=11111111    DS=0010    ESI=55555555    FS=0038
EBX=33333333    ES=0010    EDI=77777777    GS=0030
ECX=22222222    SS=0018    ESP=000003d0    EBP=00000400
EDX=00000000    CS=0008    EIP=0000045e
```

Tehát a hibakód jelzi, hogy a 100h szelektor használata okozta a 13. kivételt, a CS:EIP alapján pedig pontosan azonosítható a hibás utasítás is.

2.2. Taszkváltás egy 0. szintű taszkra

Ebben a példában átkapcsolunk egy másik taszkra, amely kiír egy karaktert a képernyőre a Screen szelektor segítségével, majd visszaugrik az előző taszkra.

Az új taszknak saját kódszegmense, adatszegmense és veremszegmense lesz, ezáltal teljesen elkülöníthető egymástól a két taszk. Az egyes taszkok által használt szelektorok a 8. ábrán láthatóak.

a szegmens típusa	0. Taszk szelektorai	1. Taszk szelektorai
kódszegmens	Code32	Code_1
adatszegmens	Data32	Data_1
veremszegmens	Stack32	Stack_1
taszk-állapotszegmens	Task_0	Task_1
képernyőszegmens	Screen	Screen

2.5. ábra A taszkok és szelektoraik

Először a GDT-ben definiálni kell két taszk-állapotszegmens deszkriptort, az egyiket a régi taszknak (Task_0), és a másikat az új taszknak, amit majd meghívunk (Task_1).

A TSS hossza most a minimális 104 byte, mert nem használjuk az I/O bittérképet. Mivel a taszk-állapotszegmenset is gyakran használjuk majd, érdemes ezt a deszkriptortípust is külön definiálni.

defs.inc

```
TSS_Desc Equ                    Present+DPL0+Sys_Desc+TSS_Desc
TSS_Len  Equ                    104
```

A GDT tehát kiegészül két taszk-állapotszegmenssel, illetve az új taszk kód-, adat-, és veremszegmens deszkriptorával. A taszk egyes szegmensei a T1_Code, a T1_Data, és a T1_Stack lesznek, illetve ezeknek a szegmenseknek a pontos hosszait pedig a T1_Code_Size, a T1_Data_Size, és a T1_Stack_Size szimbólumok tartalmazzák.

gdt.inc

```
GDT_Seg Segment

GDT Label Qword

Descriptor NullSel,0,0,0
Descriptor Code32, 0,PM_Code_Size, Code_Desc+Seg_32
Descriptor Data32, 0,PM_Data_Size, Data_Desc+Seg_32
Descriptor Stack32,0,PM_Stack_Size,Data_Desc+Seg_32
Descriptor Code16, 0,0ffffh,       Code_Desc+Seg_16
Descriptor Data16, 0,0ffffh,       Data_Desc+Seg_16
Descriptor Screen, 0b8000h,4000,   Data_Desc+Seg_32
Descriptor Flat32, 0,0fffffh,Data_Desc+Seg_32+Seg_Big

Descriptor Task_0, 0,TSS_Len,      TSS_Desc
Descriptor Task_1, 0,TSS_Len,      TSS_Desc

Descriptor Code_1, 0,T1_Code_Size, Code_Desc+Seg_32
Descriptor Data_1, 0,T1_Data_Size, Data_Desc+Seg_32
Descriptor Stack_1,0,T1_Stack_Size,Data_Desc+Seg_32

GDT_Seg Ends
```

A TSS-ben lévő regiszterek, és egyéb mezők megcímzését megkönnyíti az alábbi struktúra:

defs.inc

```
TSS_Struc Struc

TSS_BACKLINK   DW 0,0
TSS_ESP0       DD 0
TSS_SS0        DW 0,0
TSS_ESP1       DD 0
TSS_SS1        DW 0,0
TSS_ESP2       DD 0
TSS_SS2        DW 0,0
TSS_CR3        DD 0
TSS_EIP        DD 0
TSS_EFLAGS     DD 0
TSS_EAX        DD 0
TSS_ECX        DD 0
```

```
TSS_EDX         DD 0
TSS_EBX         DD 0
TSS_ESP         DD 0
TSS_EBP         DD 0
TSS_ESI         DD 0
TSS_EDI         DD 0
TSS_ES          DW 0,0
TSS_CS          DW 0,0
TSS_SS          DW 0,0
TSS_DS          DW 0,0
TSS_FS          DW 0,0
TSS_GS          DW 0,0
TSS_LDTR        DW 0,0
TSS_TBIT        DW 0
TSS_IOBASE      DW 0

TSS_Struc Ends
```

Ha a két taszk állapotszegmensét külön szegmensekbe rakjuk, akkor a bennük lévő értékek eléréséhez a TSS-ekre mutató két új adatszegmens-deszkriptort kellene definiálni. Bár ez biztonságosabb megoldás lenne, mégis célszerűbb a két TSS-t a PM_Data szegmensben elhelyezni, mivel így elérhetővé válik a Data32 szelektor segítségével.

```
PM_Data Segment

TSS_0 TSS_Struc <>
TSS_1 TSS_Struc <>

PM_Data Ends
```

A TSS deszkriptorok báziscímét is fizikai címmé kell alakítani, viszont itt már nem elég a valós módú szegmenset felhasználni, mivel a TSS-ek nem egy adott szegmens elején helyezkednek el. Tehát a báziscímeket a PM_Data szegmenscíme és a TSS-ek ofszetcíme együttesen határozza meg. A legegyszerűbb a Seg2Base makrót úgy módosítani, hogy ha szükséges, akkor a szegmens mellett egy ofszetcímet is meg lehessen adni paraméterként. Viszont hogy ne kelljen mindig az ofszetcímet is feltüntetni (általában 0-t), ezért a makróban a báziscímhez csak akkor adjuk hozzá a második paramétert, ha az nem üres (IF Not Blank).

defs.inc

```
Seg2Base Macro RealSeg,Off
   xor  eax,eax
   mov  ax,RealSeg              ; szegmens->32 bites báziscím
```

```
    shl   eax,4

    ifnb <Off>                  ; ha van ofszetcím,
    add   eax,offset Off; adjuk hozzá azt is
    endif

    add   di,2
    stosw                       ; a Bázis 0-15 bitek tárolása
    shr   eax,16                ; a felsõ 16 bit az alsó 16-ba
    stosb                       ; a Bázis 16-23 bitek tárolása
    add   di,2
    mov   al,ah
    stosb                       ; a Bázis 24-31 bitek tárolása
Endm
```

Tehát már csak a GDT-ben lévő deszkriptorok báziscímének a beállítása, illetve a GDTR inicializálása van hátra.

```
RM_Seg Segment

InitGDT Proc


    mov   di,GDT_Seg            ; ES:DI a GDT-re mutasson
    mov   es,di
    lea   di,GDT_Seg:GDT

    add   di,8                  ; a nulla deszkriptor átugrása

    Seg2Base PM_Code            ; Code32
    Seg2Base PM_Data            ; Data32
    Seg2Base PM_Stack           ; Stack32
    Seg2Base RM_Seg             ; Code16
    Seg2Base RM_Seg             ; Data16

    add   di,16                 ; a Screen és a Flat32
                                ; átugrása

    Seg2Base PM_Data,TSS_0      ; Task_0
    Seg2Base PM_Data,TSS_1      ; Task_1

    Seg2Base T1_Code            ; Code_1
    Seg2Base T1_Data            ; Data_1
    Seg2Base T1_Stack           ; Stack_1

    mov   eax,GDT_Seg
    shl   eax,4
    add   GDT_Base,eax          ; a GDT_Base fizikai címmé
                                ; alakítása
```

```
    lgdt GDT_Value              ; a GDTR beállítása

    ret

InitGDT Endp

RM_Seg Ends
```

A taszkváltáshoz egy távoli ugrást használunk majd fel, hasonlóan a szegmensek közötti ugrásokhoz. Most viszont a kódszegmens szelektora helyére a TSS szelektora kerül, és az ofszetcím pedig lényegtelen, mert úgyis a TSS-ben levő CS:EIP címtől kezdődik az utasítás-végrehajtás.

A taszkváltás előtt még be kell állítani az új taszk szegmensregisztereit. Jelen esetben a CS a kódszegmensre, a DS, és az ES az adatszegmensre, az FS a teljes 4 GB méretű lineáris memóriára, a GS a képernyőszegmensre, és az SS pedig a veremszegmensre mutat. Ezeken kívül az TSS-ben tárolni kell a verem ofszetcímét (tetejét) az ESP-ben, és a kezdeti állapotbiteket is. Az EFLAGS értékét most egyszerűen 0-ra állítjuk, vagyis a megszakítások le lesznek tiltva (IF = 0), a taszk IOPL szintje 0. lesz, a stringműveletek címei növekedni fognak (DF = 0), stb. Most megkönnyíti a szükséges értékek beállítását, hogy a két taszk-állapotszegmens a PM_Data szegmensen belül van, így a TSS-ek a DS-el elérhetőek, nem kell egy külön deszkriptort használni erre a célra.

A taszkregiszterbe még be kell tölteni a Task_0 szelektorát, ez határozza meg, hogy az első taszkváltáskor a régi taszk regiszterei melyik TSS-ben tárolódnak.

pm.inc

```
PM_Code Segment Para
        Assume cs: PM_Code, ds:PM_Data, es:PM_Data

PM_Main:

    lea  esi,Ex2_Str
    call WriteStr

    mov  ax,Task_0              ; az elsõ taszk kiválasztása
    ltr  ax

    lea  edx,TSS_1              ; DS:EDX = az új taszk TSS-e

    mov  [edx].TSS_CS,Code_1
```

```
   mov  [edx].TSS_EIP,0

   mov  [edx].TSS_DS,Data_1
   mov  [edx].TSS_ES,Data_1
   mov  [edx].TSS_FS,Flat32
   mov  [edx].TSS_GS,Screen

   mov  [edx].TSS_SS,Stack_1
   mov  [edx].TSS_ESP,T1_Stack_Size

   mov  [edx].TSS_EFLAGS,0

   Jump32_32 Task_1,0          ; ugrás az új taszkra

   ret

PM_Code Ends


PM_Data Segment

Ex2_Str  db '2. Példa: Taszkkapcsolás',13,13
         db 'Az új taszk egy pontot ír majd ide '
         db '----> [ ]',13,0

PM_Data Ends
```

Az Ex2_Str stringben lévő két szögletes zárójel közötti üres hely a képernyőn pontosan az 5. sor 42. oszlopában helyezkedik el, vagyis az új taszknak majd ide kell egy pontot raknia. A karaktert és az XY koordináta értékét az adatszegmensben helyezzük el, hogy az új taszk adatszegmens szelektorát is leellenőrizzük, továbbá a taszk saját vermét is teszteljük.

task.inc

```
T1_Code Segment Para Public Use32
        Assume cs: T1_Code, ds: T1_Data
T1_Code_Start Label Byte

   mov  ax,Char                ; egy fehér pont kiírása
   mov  edx,XY
   mov  gs:[edx],ax            ; a megfelelõ helyre

   push eax                    ; a verem kipróbálása
   pop  eax

   Jump32_32 Task_0,0          ; visszaugrás az elõzõ taszkra

T1_Code_Size Equ $-T1_Code_Start
T1_Code Ends
```

```
T1_Data Segment Para Public Use32
T1_Data_Start Label Byte

Char    dw '.'+15*256          ; egy fehér színû pont
XY dd 5*160+42*2               ; 5. sor 42. oszlop címe

T1_Data_Size Equ $-T1_Data_Start
T1_Data Ends

T1_Stack Segment Para Stack
T1_Stack_Size Equ 400h
db T1_Stack_Size dup (?)       ; külön verem a taszk számára
T1_Stack Ends
```

2.3. Taszkváltás egy 3. szintű taszkra

Ahhoz, hogy az előző példához hasonlóan a képernyőre írást egy 3. privilégiumszintű taszkból végezzük el, a következő feltételek szükségesek:

- a 3. szintről is elérhető TSS deszkriptor, vagy taszkkapu
- 3. szintű kód-, adat-, és veremszegmens
- 0. (esetleg 1. és 2.) szintű veremszegmens deszkriptor
- a képernyőre mutató 3. szintű deszkriptor
- megszakítások és kivételek a 3. szintről is hívhatóak legyenek

A taszkkapuk használata akkor ajánlott, amikor azt akarjuk, hogy egyes taszkok elérjenek bizonyos TSS deszkriptorokat, viszont ebben más taszkokat meg akarunk akadályozni. Ez a korlátozás csak akkor igazán hatékony, ha a taszkkapuk lokálisak, vagyis az LDT-ben szerepelnek, hiszen a GDT-ben lévő taszkkapukat bármelyik azonos szintű taszk felhasználhatja. Mivel most csak egy darab 3. szintű taszk lesz, ezért most még a TSS deszkriptort használjuk közvetlenül a taszkváltásokra.

Célszerű külön definiálni a 3. privilégiumszintű kód-, adat-, és a megszakításkapu illetve a TSS deszkriptorok típusát a 0. szintűekhez hasonlóan.

defs.inc

```
Code_Desc3     Equ             Present+DPL3+Seg_Desc+Code_ER
Data_Desc3     Equ             Present+DPL3+Seg_Desc+Data_RW

TSS_Desc3      Equ             Present+DPL3+Sys_Desc+Sys_TSS
IntGate_Desc3  Equ             Present+DPL3+Sys_Desc+Sys_IntGate
```

Ezek felhasználásával a GDT-ben az új taszkhoz tartozó (eddig 0. szintű) deszkriptorokat módosítani kell, hogy 3. szintűek legyenek, és ezen kívül létre kell hozni egy új adatdeszkriptort a taszk 0. szintű veremhez. Most csak a 0. és a 3. szintet használjuk, ezért az 1. és a 2. szintek számára nem szükséges külön verem kialakítása. Továbbá ahhoz, hogy a videomemória a 3. szintről is elérhető legyen, a GDT-ben definiáljuk a Screen3 deszkriptort is.

gdt.inc

```
Descriptor Task_0, 0,TSS_Len,       TSS_Desc3
Descriptor Task_1, 0,TSS_Len,       TSS_Desc3

Descriptor Code_1, 0,T1_Code_Size, Code_Desc3+Seg_32
Descriptor Data_1, 0,T1_Data_Size, Data_Desc3+Seg_32
Descriptor Stack_1,0,T1_Stack_Size,Data_Desc3+Seg_32
Descriptor Stack0_1,0,T1_Stack0_Size,Data_Desc+Seg_32
Descriptor Screen3, 0b8000h,4000,  Data_Desc3+Seg_32
```

A szegmensregiszterekbe viszont nem szabad közvetlenül a fenti szelektorokat betölteni, mert ekkor a taszkváltáskor a 10. kivétel (érvénytelen TSS) keletkezne. Ennek oka, hogy a CS-be kerülő szelektor igényelt privilegizálási szintje (RPL) és az adott kódszegmens deszkriptor privilégiumszintje (DPL) nem egyezik meg. De mindez nemcsak a CS-re vonatkozik, hanem az összes többi szegmensregiszter esetén is hasonló a helyzet.

Az alábbi 4 szimbólum az igényelt privilegizálási szintek értékeit definiálják, tehát amit hozzá kell adni az egyes szelektorokhoz.

defs.inc

```
RPL0     Equ  0                ; 0. szint
RPL1     Equ  1                ; 1. szint
RPL2     Equ  2                ; 2. szint
RPL3     Equ  3                ; 3. szint
```

Az IDT-ben lévő deszkriptorokat is át kell alakítani ahhoz, hogy a 3. szintről is elérhetőek legyenek. Ezért az eddigi 0. szintű kapukat létrehozó Int_Gate makró helyett az alábbi Int_Gate3 makrót használjuk fel a megszakítások definiálására.

```
Int_Gate3 MACRO Num,Sel,Off
   org   IDT+Num*8
   dw    small Off
   dw    Sel
```

```
    db    0
    db    IntGate_Desc3          ; 3. szintû megszakításkapu
    dw    0
Endm
```

A változatosság kedvéért ebben a példában nem a FAR JMP-vel ugrunk az új taszkra, hanem meghívjuk egy FAR CALL utasítással. Ehhez szükség van egy új makróra, amelynek paramétere a Jmp32_32 makróhoz hasonlóan a cél szelektora és ofszetcíme.

```
CallFar Macro Sel,Off
    db 09ah
    dd offset Off
    dw Sel
    Endm
```

A 3. és a 0. szintű TSS inicializálása abban különbözik egymástól, hogy itt be kell állítani a 0. szintű vermet is (SS0:ESP0), a szelektorokban megváltozott az RPL értéke, illetve hogy a GS-be egy 3. szintű deszkriptor kerül, amely a képernyőre mutat. Az FS-be is a taszk adatszegmens szelektora kerül, mivel egyáltalán nem biztonságos egy alkalmazói szinten lévő program számára a teljes lineáris memória elérését (írás/olvasás) lehetővé tenni. (Bár az is igaz, hogy a Flat32 0. szintű, tehát a 3. szintű taszk nem használhatja.)

pm.inc

```
PM_Code Segment Para
        Assume cs: PM_Code, ds:PM_Data, es:PM_Data
PM_Main:

    lea   esi,Ex3_Str
    call  WriteStr

    mov   ax,Task_0              ; az elsõ taszk kiválasztása
    ltr   ax

    lea   edx,TSS_1              ; DS:EDX = az új taszk TSS-e

    mov   [edx].TSS_CS,Code_1+RPL3
    mov   [edx].TSS_EIP,0

    mov   [edx].TSS_DS,Data_1+RPL3
    mov   [edx].TSS_ES,Data_1+RPL3
    mov   [edx].TSS_FS,Data_1+RPL3
```

```
    mov   [edx].TSS_GS,Screen3+RPL3

    mov   [edx].TSS_SS,Stack_1+RPL3
    mov   [edx].TSS_ESP,T1_Stack_Size
    mov   [edx].TSS_SS0,Stack0_1
    mov   [edx].TSS_ESP0,T1_Stack0_Size

    mov   [edx].TSS_EFLAGS,0

    CallFar Task_1,0            ; a 3. szintû taszk meghívása

    ret

PM_Code Ends


PM_Data Segment

Ex3_Str db '3. Példa: Egy 3. privilégiumszintû taszk'
        db ' meghívása.',13,13
        db 'Az új taszk egy pontot ír majd ide '
        db '----> [ ]',13,0

PM_Data Ends
```

A meghívott taszk tehát a kapcsos zárójelek közé kiír egy fehér színű pontot. A két veremszegmenset is ajánlatos tesztelni, hogy jól működnek-e. A 3. szintű vermet egyszerű push/pop utasításpárral kipróbálhatjuk, a 0. szintű verem működését pedig a töréspont kivétellel ellenőrizhetjük le. Mivel a 3. kivétel megszakításkapuja egy 0. szintű kódszegmensre mutat, ezért a megszakítás hívásakor a processzor automatikusan átkapcsol a biztonságosabb 0. szintű veremre.

A taszkból az IRETD utasítással léphetünk ki, amely észleli, hogy a CALL miatt az NT bit értéke 1, és ezért a taszk-állapotszegmens BackLink mezőben lévő taszknak (T0) adja vissza a vezérlést.

task.inc

```
T1_Code Segment Para Public Use32
       Assume cs: T1_Code, ds: T1_Data
T1_Code_Start Label Byte

    mov   ax,Char               ; egy fehér pont kiírása
    mov   edx,XY
    mov   gs:[edx],ax           ; a megfelelõ helyre

    push eax                    ; a 3. szintû verem használata
```

```
    pop  eax
    int  3                      ; a 0. szintû verem használata
    iretd                       ; visszatérés a hívó taszkra
T1_Code_Size Equ $-T1_Code_Start
T1_Code Ends

T1_Data Segment Para Public Use32
T1_Data_Start Label Byte

Char    dw '.'+15*256           ; egy fehér színû pont
XY dd 5*160+42*2                ; 5. sor 42. oszlop címe

T1_Data_Size Equ $-T1_Data_Start
T1_Data Ends

T1_Stack Segment Para Stack  ; 3. szintû verem
T1_Stack_Size Equ 400h
db T1_Stack_Size dup (?)
T1_Stack Ends

T1_Stack0 Segment Para Stack ; 0. szintû verem
T1_Stack0_Size Equ 400h
db T1_Stack0_Size dup (?)
T1_Stack0 Ends
```

A regiszterek kiírásakor jól látható, hogy a szegmensregiszterekben 3. szintű szelektorok vannak (az utolsó számjegy 3 illetve B), kivéve az SS-t, amelyik a T1 taszk 0. szintű vermének szelektorát (Stack0_1, 68h) tartalmazza. Mivel a 3. kivétel egy trap, a veremben lévő CS:EIP az INT 3 utáni IRETD utasításra mutat.

```
3. kivétel: töréspont

EAX=00000f2e    DS=005b    ESI=00000000    FS=005b
EBX=00000000    ES=005b    EDI=00000000    GS=0073
ECX=00000000    SS=0068    ESP=000003d0    EBP=00000000
EDX=00000374    CS=0053    EIP=00000013
```

2.4. Funkciók hívása megszakításokon keresztül.

Az előbbi példában a 3. szintű taszk a Screen3 szelektor segítségével írt a képernyőre. Viszont nem mindig tanácsos biztosítani az alkalmazói programoknak, hogy közvetlenül elérjék a képernyőt. Ha például egyszerre 10 taszk próbál egymástól függetlenül ugyanarra a képernyőre írni, akkor az könnyen káoszhoz vezethet.

Most megszüntetjük a Screen3 szelektort, és a GS-be is a taszk adatszegmense kerül, így a 3. szintű taszk nem írhat közvetlenül a videomemóriába. Ehelyett létrehozunk egy megszakításkaput, amelyen keresztül a taszk már meghívhat bizonyos képernyőkezelő funkciókat.

Az első 32 darab megszakítást továbbra is meghagyjuk a kivételek számára, ezek után a 20h–2Fh között most a hardver megszakítások vannak, ezért a funkcióhívásokhoz a következőt, az INT 30h-t fogjuk használni. Ahhoz, hogy ez a 3. szintről is meghívható legyen, a megszakításkapu deszkriptorában a DPL-t háromra kell állítani, vagyis a kaput az Int_Gate3 makróval hozzuk majd létre az IDT-ben. A megszakítás-kezelőt a 0. szintű kódszegmensben (PM_Code) helyezzük el, ezáltal a megszakítás alatt az aktuális privilégiumszint (CPL) is 0 lesz, mivel ezt a CS határozza meg. Így a meghívott képernyőkezelő eljárások már használhatják a 0. szintű Screen szelektort is.

idt.inc

```
IDT Label Qword
    ...
    Int_Gate3 30h,Code32,PM_Code:Int_30
```

Az INT 30h most csak három képernyőkezelő funkciót fog tartalmazni, a karakter és számok kiírását, illetve a kurzorpozícionálást. Természetesen a többit is be lehetne állítani (képernyőtörlés, stb.), de a későbbiekben csak ezekre lesz szükségünk. A funkció sorszámát az AL regiszter tartalmazza, a paramétert pedig az EDX.

Funkció	Sorszám	Paraméter
Karakter kiírása	AL=0	DL=karakter
Szám kiírása	AL=1	EDX=szám
XY beállítása	AL=2	DH=sor, DL=oszlop

2.6. ábra Az INT 30h funkciói

Több funkció esetén célszerű egy táblázatba foglalni a funkciók címeit, és a sorszámnak megfelelő eljárást meghívni, mivel ez sokkal gyorsabb módszer. Ezt a táblázatot akár a kódszegmensben is tárolhatjuk, ha az olvasható és végrehajtható is.

```
PM_Code Segment

Int_Handler:

    push eax

    movzx eax,al                ; EAX = funkció
    cmp  eax,LastFunc           ; érvényes funkció?
    ja   Done                   ; nem.

    shl  eax,2                  ; *4
    call cs:[eax+FuncTable]     ; a funkció hívása

    pop  eax
    iretd                       ; megszakítás-kezelõ vége

FuncTable Label Dword           ; az eljárások táblázata
    dd offset F_1
    dd offset F_2
    dd offset F_3
    .
    .

LastFunc=$-FuncTable/4          ; az érvényes funkciók száma

F_1:                            ; 1. funkció kezelõje
    ...
    ret
F_2:                            ; 2. funkció kezelõje
    ...
    ret

    .
    .
    .

PM_Code Ends
```

Mivel most csak néhány funkciót tartalmaz az INT 30h, ezért használhatjuk a hagyományos ellenőrzést is.

pm.inc

```
PM_Code Segment

Int_30 Proc

   push eax ds

   push Data32                ; DS = 0. szintû adatszegmens
   pop  ds
F_0:
   cmp  al,0                  ; 0. funkció?
   jne  F_1

   mov  al,dl
   call WriteChar             ; karakter kiírása
   jmp  F_End
F_1:
   cmp  al,1                  ; 1. funkció?
   jne  F_2

   call WriteHex              ; hexadecimális szám kiírása
   jmp  F_End

F_2:
   cmp  al,2                  ; 2. funkció?
   jne  F_End

   mov  ScreenX,dl            ; az XY koordináták beállítása
   mov  ScreenY,dh

F_End:
   pop  ds eax
   iretd

Int_30 Endp

PM_Code Ends
```

A hosszabb ideig tartó funkcióhívásoknál a megszakításokat is engedélyezni kell, hogy a hardver eszközöket helyesen, késedelem nélkül lehessen kezelni. Másik megoldás erre a trapkapuk használata, amelyek nem befolyásolják az IF bitet.

A meghívott 3. szintű taszk kiír egy hexadecimális számot a képernyő egy bizonyos helyére, az előző példától eltérően tehát most nem a videomemóriába, hanem az INT 30h megszakításon keresztül.

```
T1_Code Segment Para Public Use32
        Assume cs: T1_Code, ds: T1_Data
T1_Code_Start Label Byte

    mov  edx,0537h              ; az XY koordináták (5,55)
    mov  al,2                   ; 2. funkció
    int  30h                    ; megszakítás hívása

    mov  edx,12345678h          ; a teszt szám
    dec  al                     ; 1. funkció
    int  30h                    ; megszakítás hívása

    iretd                       ; visszatérés a hívó taszkra

T1_Code_Size Equ $-T1_Code_Start
T1_Code Ends
```

Bár a képernyőkezelést nem kell mindenképpen kapukon keresztül megoldani, de bizonyos dolgokat – például a memóriakezelést, a fileműveleteket, és egyéb driverek hívását – célszerű felsőbb szintű hívásokon keresztül elérhetővé tenni az alkalmazói programok számára.

2.5. Két darab 3. privilégiumszintű taszk váltogatása

Ebben a példában a 0. szintű taszk két különálló, 3. szintű taszkot hív meg sokszor egymás után. Ezek a taszkok, amikor megkapják a vezérlést, megnövelnek egy számlálót, és kiírják ennek az aktuális értékét, majd visszaugranak a 0. szintű taszkra.

Az első taszkhoz hasonlóan a GDT-ben létre kell hozni a második taszk számára is külön kód-, adat-, és veremszegmens deszkriptorokat, illetve egy taszk-állapotszegmenset.

gdt.inc

```
GDT Label Qword

...
```

```
; Taszk-állapotszegmensek
Descriptor Task_0,0,TSS_Len,TSS_Desc3
Descriptor Task_1,0,TSS_Len,TSS_Desc3
Descriptor Task_2,0,TSS_Len,TSS_Desc3

; 1. taszk deszkriptorai
Descriptor Code_1,   0,T1_Code_Size,   Code_Desc3+Seg_32
Descriptor Data_1,   0,T1_Data_Size,   Data_Desc3+Seg_32
Descriptor Stack_1,  0,T1_Stack_Size,  Data_Desc3+Seg_32
Descriptor Stack0_1,0,T1_Stack0_Size,Data_Desc+Seg_32

; 2. taszk deszkriptorai
Descriptor Code_2,   0,T2_Code_Size,   Code_Desc3+Seg_32
Descriptor Data_2,   0,T2_Data_Size,   Data_Desc3+Seg_32
Descriptor Stack_2 ,0,T2_Stack_Size,  Data_Desc3+Seg_32
Descriptor Stack0_2,0,T2_Stack0_Size,Data_Desc+Seg_32
```

Nem szabad elfelejteni, hogy védett módba kapcsolás előtt a deszkriptorok báziscímét is be kell állítani. Az InitGDT eljárásban szereplő szegmens-átalakítások sorrendjének pontosan meg kell egyeznie a deszkriptortáblázatban lévő bejegyzések sorrendjével, különben a deszkriptorok báziscíme helytelen lesz.

```
InitGDT Proc

    ...

    Seg2Base PM_Data,TSS_0    ; Task_0
    Seg2Base PM_Data,TSS_1    ; Task_1
    Seg2Base PM_Data,TSS_2    ; Task_2

    Seg2Base T1_Code          ; Code_1
    Seg2Base T1_Data          ; Data_1
    Seg2Base T1_Stack         ; Stack_1
    Seg2Base T1_Stack0        ; Stack0_1

    Seg2Base T2_Code          ; Code_2
    Seg2Base T2_Data          ; Data_2
    Seg2Base T2_Stack         ; Stack_2
    Seg2Base T2_Stack0        ; Stack0_2

    ...

InitGDT Endp
```

A taszkkapcsolások előtt inicializálni kell mindkét taszk TSS-ét a megfelelő szelektorokkal, és egyéb értékekkel. Ezek után a 0. szintű taszk

„néhány" alkalommal meghívja a másik két taszkot egymás után, amelyek kiírják a saját számlálójukat a szokásos kapcsos zárójelek közé.

pm.inc

```
PM_Code Segment

PM_Main:

    lea  esi,Ex5_Str
    call WriteStr

    mov  ax,Task_0              ; az elsõ taszk kiválasztása
    ltr  ax

    lea  edx,TSS_1              ; DS:EDX = az 1. taszk TSS-e

    mov  [edx].TSS_CS,Code_1+RPL3

    ...

    lea  edx,TSS_2              ; DS:EDX = a 2. taszk TSS-e

    mov  [edx].TSS_CS,Code_2+RPL3
    mov  [edx].TSS_EIP,0

    mov  ax,Data_2+RPL3
    mov  [edx].TSS_DS,ax
    mov  [edx].TSS_ES,ax
    mov  [edx].TSS_FS,ax
    mov  [edx].TSS_GS,ax

    mov  [edx].TSS_SS,Stack_2+RPL3
    mov  [edx].TSS_ESP,T2_Stack_Size
    mov  [edx].TSS_SS0,Stack0_2
    mov  [edx].TSS_ESP0,T2_Stack0_Size

    mov  [edx].TSS_EFLAGS,0

    mov  ecx,10000h             ; kb. 260 ezer taszkváltás
PM_1:
    CallFar Task_1,0            ; az 1. taszk meghívása
    CallFar Task_2,0            ; a 2. taszk meghívása

    loop PM_1

    ret

PM_Code Ends

PM_Data Segment
```

```
Ex5_Str db '5. Példa: Két taszk váltogatása.',13,13
        db '1. Taszk számlálója:  [          ]',13
        db '2. Taszk számlálója:  [          ]',13,0

PM_Data Ends
```

A két taszkot tehát egy FAR CALL-al hívjuk meg, és az IRETD utasítással térünk vissza. Az első hívásnál ezzel még nincs is baj. Viszont amikor másodszor kapcsolunk át ugyanarra a taszkra, akkor az előző IRETD utáni utasítás hajtódik végre, amely nem feltétlenül értelmes utasítás, vagy esetleg éppen ott van a kódszegmens vége, amitől általános védelmi hiba keletkezik. Hogy ne kelljen minden CALL előtt az adott taszk TSS-ében az EIP értékét 0-ra állítani, ezért az IRETD után egy ugró utasítást helyezünk el, amely a kódszegmens elejére mutat.

A T1 és a T2 taszkok szinte csak a szegmensek nevében különböznek egymástól, illetve abban, hogy a T1 taszk az 5. sor 24. oszlopába írja ki a számláló tartalmát (a kapcsos zárójelek közé), a T2 pedig a 6. sorba.

```
T1_Code Segment Para Public Use32
        Assume cs: T1_Code, ds: T1_Data
T1_Code_Start Label Byte

L1:
    mov  edx,0518h              ; az XY koordináták (5,24)
    mov  al,2                   ; 2. funkció
    int  30h                    ; megszakítás hívása

    mov  edx,T1_Counter; a számláló kiírása
    dec  al                     ; 1. funkció
    int  30h                    ; megszakítás hívása

    inc  T1_Counter             ; a számláló növelése

    iretd                       ; visszatérés a hívó taszkra

    jmp  L1

T1_Code_Size Equ $-T1_Code_Start
T1_Code Ends

T1_Data Segment Para Public Use32
T1_Data_Start Label Byte

T1_Counter dd 0

T1_Data_Size Equ $-T1_Data_Start
T1_Data Ends
```

2.5.1. A globális deszkriptorok veszélyei

Normális esetben a 0. szintű taszk gond nélkül kapcsolgat a másik két taszk között. De például a T2 taszk bármikor „elhatározhatja", hogy amit csak tud, mindent megsemmisít.

A T2-nek egyszerűen csak végig kell menni a GDT-ben lévő 8192 darab deszkriptoron (0, 8, 10h, 18h, stb. szelektorok), és a LAR utasítás segítségével meg kell vizsgálnia a deszkriptorok elérési jogát. Célszerű például 3. szintű, jelenlévő, írható és olvasható adatszegmenseket keresni, ezeknél az elérési jogban a 8–15. bitek értéke 0F2h illetve 0F3h, az A (Accessed) bittől függően, vagyis hogy használták-e már az adott deszkriptort, vagy nem. Ide tartoznak azok a veremszegmensek is, amelyek nem lefelé bővülnek. Ennél még egyszerűbb, ha a VERW utasítást használjuk, amellyel kideríthető az összes olyan szegmens, amelybe írni tudunk. Ha sikerült találni ilyen adatszegmens deszkriptort, akkor máris kezdődhet a pusztítás:

```
; ES = egy másik 3. szintû taszk adatszegmense

    lsl   ecx,es                ; ECX=az adott szegmens hossza
    shr   ecx,2                 ; dupla szavakban
    mov   eax,'HEHE'            ;-)
    xor   edi,edi
    rep   stosd                 ; vajon túléli a másik taszk?
```

És ezt meg lehet ismételni az összes deszkriptoron, amelyeket elér a taszk. A „gonosz" taszknak azt is ellenőriznie kell, hogy a kiszemelt deszkriptor ne a saját adat-, vagy veremszegmense legyen, mert akkor öngyilkosságot követne el. A kódszegmens felülírása ebből a szempontból sokkal eredményesebb lenne, ha védett módban is lehetne a kódszegmensekbe írni, de azért még így is lehetőség van a többi taszk kódjának kiszimatolására. A taszk számára elérhető többi TSS deszkriptorral ugyancsak vissza lehet élni.

Mindezeket akár ki is próbálhatjuk, és kiírhatjuk a képernyőre a potenciális veszélyt hordozó globális deszkriptorokat. Az Ex5_Str kis módosításával elérhető, hogy ezek a deszkriptorok jól láthatóak legyenek. Az írható adatszegmenseket az üresen hagyott 9. sorba, a TSS-eket pedig a 11. sorba írjuk majd ki.

```
PM_Data Segment

Ex5_Str db '5. Példa: Két taszk váltogatása.',13,13
        db '1. Taszk számlálója:  [          ]',13
        db '2. Taszk számlálója:  [          ]',13,13
        db 'T2 által elérhetõ adatszegmensek: ',13,13
        db 'T2 által elérhetõ TSS-ek: ',0

PM_Data Ends
```

A következő Find_Data eljárás végignézi az összes GDT-ben lévő deszkriptort, és olyanokat keres, amelyeket esetleg a T2 taszk meg tudna rongálni (vagyis a VERW után Z = 1), majd ezeket kiírja egymás után a képernyőre egy szóközzel elválasztva.

task.inc

```
T2_Code Segment

Find_Data:
   mov  edx,0900h              ; 9. sor eleje
   mov  al,2                   ; 2. funkció
   int  30h                    ; megszakítás hívása

   mov  ecx,8192               ; deszkriptorok száma
   xor  edx,edx                ; 0-tól kezdve

FD_Check:
   verw dx                     ; írás ellenõrzése
   jz   FD_Show                ; írható szegmens

FD_Next:
   add  dx,8                   ; következõ deszkriptor
   loop FD_Check
   ret

FD_Show:
   mov  al,1                   ; szelektor kiírása
   int  30h                    ; megszakítás hívása

   push edx
   mov  dl,' '
   dec  al
   int  30h                    ; egy szóköz kiírása
   pop  edx

   jmp  FD_Next

T2_Code Ends
```

A T2 taszk a mostani GDT alapján tehát a 60h, 68h, 80h és 88h (Data_1, Stack_1, Data_2, és Stack_2) szelektorok használatával (+RPL3) tud írni az adott szegmensekbe, vagyis a saját és a T1 taszkhoz tartozó adatszegmensekbe.

A taszk-állapotszegmensek is veszélyt rejtenek magukban, mivel a TSS deszkriptorok csak a GDT-ben lehetnek. Eddig a DPL-t háromra kellett állítani, hogy egyes alkalmazói programok is egy másik taszkra ugorhassanak. Viszont a 3. szintű TSS deszkriptorokat nem csak azok a taszkok érik el, amelyeknek szabadna, hanem az összes többi is. Most a T0 hívja meg a T1 és a T2 taszkokat, de például a T2 taszk is bármikor a T1-re ugorhat.

A TSS-ek felkutatását most a LAR utasítással végezzük el, és jelenlévő, 3. szintű, szabad (B = 0) vagy foglalt (B = 1) TSS-eket keresünk. A LAR utasítás végrehajtása után a Z flag 1 lesz, ha az elérési jog tárolása sikerült. Ha a betöltött elérési jog egy taszk-állapotszegmenset jelöl, akkor az aktuális deszkriptort kiírjuk a 11. sorba.

```
T2_Code Segment

Find_TSS:
    mov   edx,0b00h             ; 11. sor eleje
    mov   al,2                  ; 2. funkció
    int   30h                   ; megszakítás hívása

    mov   ecx,8192              ; deszkriptorok max. száma
    xor   edx,edx               ; 0-tól kezdve

FT_Check:
    lar   ax,dx
    jnz   FT_Next               ; ez nem elérhetõ

    cmp   ah,0e9h               ; P=1, DPL=3, S=0, típus=9
    je    FT_Show
    cmp   ah,0ebh               ; P=1, DPL=3, S=0, típus=11
    je    FT_Show

FT_Next:
    add   dx,8                  ; következõ deszkriptor
    loop  FT_Check
    ret

FT_Show:
    mov   al,1                  ; szelektor kiírása
    int   30h                   ; megszakítás hívása
```

```
    push edx
    mov  dl,' '
    dec  al
    int  30h                    ; egy szóköz kiírása
    pop  edx

    jmp  FT_Next

T2_Code Ends
```

A Find_TSS hívásakor a Task_0, a Task_1, és a Task_2 (40h, 48h, 50h) íródik ki a képernyőre, vagyis mind a három darab TSS-t sikerült megtalálni.

A fentieknek persze nem az volt a célja, hogy ötleteket adjon, hogyan lehet könnyedén egy kevésbé biztonságos rendszerben óriási károkat okozni, hanem az, hogy felhívja a figyelmet a globális deszkriptorok használatának veszélyeire.

Hogy ezt a védelmi hiányosságot is kiküszöbölhessük, ezért a következő példában a taszkok szegmenseit lokális deszkriptortáblázatban helyezzük el.

2.6. Lokális deszkriptorok

Azért, hogy megpróbáljuk az illetéktelen hozzáféréseket megakadályozni, a következő lépéseket tesszük:

- A GDT-ben minden 3. szintű taszk számára egy külön LDT deszkriptort hozunk létre.
- A 3. szintű taszkok szegmensdeszkriptorait töröljük a GDT-ből, és a lokális deszkriptortáblázatokban helyezzük el.
- Minden TSS deszkriptor privilégiumszintjét 0-ra állítjuk, tehát így az alkalmazói szintről már nem érhetőek el.
- A taszkok saját LDT-jében 3. szintű taszkkapukat helyezünk el, amelyek csak azokra a TSS-re mutatnak, amelyek eléréséhez joguk van.

A következő két táblázatban látható az egyes taszkokhoz tartozó deszkriptorok elhelyezkedése az LDT használata előtt és után.

	T0 taszk	T1 taszk	T2 taszk
GDT 0. szint	Code32 Data32 Stack32 Screen Flat32	Stack0_1	Stack0_2
GDT 3. szint		Task_1 Code_1 Data_1 Stack_1	Task_2 Code_2 Data_2 Stack_2
LDT 0. szint			
LDT 3. szint			

2.7. ábra Csak a GDT használata

	T0 taszk	T1 taszk	T2 taszk
GDT 0. szint	Code32 Data32 Stack32 Screen Flat32	Task_1 LDT_1	Task_2 LDT_2
GDT 3. szint			
LDT 0. szint		Stack0_1	Stack0_2
LDT 3. szint		Code_1 Data_1 Stack_1 Gate_1	Code_2 Data_2 Stack_2 Gate_2

2.8. ábra A GDT és az LDT együttes használata

A globális deszkriptortáblázatot tehát most kiegészítjük két darab LDT deszkriptorral, és a TSS-ek privilégiumszintjét is visszaállítjuk úgy, hogy csak a felügyelői szintről lehessen elérni (DPL = 0).

A T0 taszk deszkriptorait nem érdemes lokális táblázatban elhelyezni, egyrészt mivel a 0. privilégiumszint miatt biztonságban vannak, másrészt a többi taszk számára elérhetetlenné válnának a mostani INT 30h funkciók. (Mivel ezek a Code32, Data32, Stack32, és a Screen szelektort használják.) Ez utóbbi áthidalható lenne, ha minden taszk LDT-jének azonos pozíciójá-

ban helyezzük el a szükséges deszkriptorokat, vagy ha a funkciók számára külön deszkriptorokat hozunk létre a GDT-ben, és a megszakításkaput az új kódszegmensre irányítjuk. De mivel a T1 és a T2 nem érheti el a T0 taszk 0. szintű deszkriptorait, ezért ezeket meghagyjuk a GDT-ben.

Az LDT deszkriptorokat az LDT_Desc szimbólum (Present + DPL0 + Sys_Desc + Sys_LDT) segítségével definiáljuk. Használhatjuk a Descriptor makrót is, bár az valójában a szegmensdeszkriptorok létrehozására szolgál, viszont az LDT deszkriptorok felépítése (bázis, határ, típus) hasonlít hozzájuk.

A módosítások után a GDT a következő deszkriptorokat tartalmazza:

gdt.inc

```
GDT_Seg Segment Para Public

GDT  Label Qword

Descriptor NullSel,0,0,0
Descriptor Code32, 0,PM_Code_Size, Code_Desc+Seg_32
Descriptor Data32, 0,PM_Data_Size, Data_Desc+Seg_32
Descriptor Stack32,0,PM_Stack_Size,Data_Desc+Seg_32
Descriptor Code16, 0,0ffffh,       Code_Desc+Seg_16
Descriptor Data16, 0,0ffffh,       Data_Desc+Seg_16
Descriptor Screen, 0b8000h,4000,   Data_Desc+Seg_32
Descriptor Flat32, 0,0fffffh,Data_Desc+Seg_32+Seg_Big

Descriptor Task_0, 0,TSS_Size,     TSS_Desc
Descriptor Task_1, 0,TSS_Size,     TSS_Desc
Descriptor Task_2, 0,TSS_Size,     TSS_Desc

Descriptor LDT_1, 0,T1_LDT_Size,   LDT_Desc3
Descriptor LDT_2, 0,T2_LDT_Size,   LDT_Desc3

GDT_Len    Equ $-GDT                ; a GDT hossza

GDT_Seg Ends
```

A lokális deszkriptortáblázatok báziscímét is meg kell változtatni, akárcsak a T0 taszk szegmensdeszkriptorainak, illetve a három darab TSS-nek a báziscímét.

```
RM_Seg Segment

InitGDT Proc

    mov  ax,GDT_Seg           ; es = a GDT szegmense
    mov  es,ax
    lea  di,GDT_Seg:GDT       ; di = a GDT címe

    add  di,8                 ; a nulla deszkriptor

    Seg2Base PM_Code          ; Code32
    Seg2Base PM_Data          ; Data32
    Seg2Base PM_Stack         ; Stack32
    Seg2Base RM_Seg           ; Code16
    Seg2Base RM_Seg           ; Data16

    add  di,16                ; a Screen és a Flat32

    Seg2Base PM_Data,TSS_0    ; Task_0
    Seg2Base PM_Data,TSS_1    ; Task_1
    Seg2Base PM_Data,TSS_2    ; Task_2

    Seg2Base PM_Data,T1_LDT   ; LDT_1
    Seg2Base PM_Data,T2_LDT   ; LDT_2

    mov  eax,GDT_Seg
    shl  eax,4
    add  GDT_Base,eax         ; a GDT_Base módosítása

    lgdt GDT_Value            ; a GDTR beállítása

    ret

InitGDT Endp

RM_Seg Ends
```

A Descriptor makró jelenleg csak arra alkalmas, hogy a GDT-ben hozzunk létre deszkriptorokat. Létrehozhatunk akár egy újabb makrót is, de talán célszerűbb a meglévőt módosítani úgy, hogy egy külön táblázatban lokális deszkriptorokat is definiálhassunk. Ezért a makrót kiegészítjük egy opcionális paraméterrel (Table), amely jelzi, hogy melyik táblázatba kerüljön az adott deszkriptor. Ha hiányzik ez az utolsó paraméter, akkor a deszkriptor a GDT-be kerül, akárcsak eddig. Ha pedig meg van adva, akkor a paraméter az aktuális LDT táblázat címét tartalmazza. Ez elsősorban a megadott szelektor értékének pontos megállapításához szükséges. A makró automatikusan beállítja a szelektorban a TI (Table Indicator) bitet is, attól

függően, hogy a deszkriptor a GDT-ben vagy az LDT-ben van-e. A rövidség kedvéért megoldjuk azt is, hogy a Desc néven is elérhető legyen a makró.

defs.inc

```
Desc Equ Descriptor

Descriptor Macro  Name,Base,Limit,Type,Table
    ifb  <Table>
    Name Equ $-GDT              ; globális deszkriptor (TI=0)
    else
    Name Equ $-Table+4          ; lokális deszkriptor (TI=1)
    endif
    dw   Limit and 0ffffh
    dw   Base and 0ffffh
    db   (Base shr 16) and 0ffh
    dw   Type or ((Limit shr 8) and 0f00h)
    db   Base shr 24
Endm
```

A két taszkhoz tartozó lokális táblázatokat (T1_LDT, T2_LDT) a taszk-állapotszegmensekhez hasonlóan a PM_Data szegmensben helyezzük el (elvileg bárhova kerülhetnének). Ezekbe a táblázatokba kerülnek az egyes taszkokhoz tartozó kód-, és adatszegmensek, illetve a 3. és 0. privilégium-szintű veremszegmensek. Az LDT-be kerül továbbá egy 3. szintű taszkkapu is, amely a T0 taszk 0. szintű TSS deszkriptorára mutat, vagyis így a T1 és a T2 már csak a T0 taszkra tudnak átkapcsolni, egymásra viszont már nem.

ldt.inc

```
PM_Data Segment

; A T1 taszk lokális deszkriptorai

T1_LDT Label Qword
Desc Code_1, 0,T1_Code_Size, Code_Desc3+Seg_32,T1_LDT
Desc Data_1, 0,T1_Data_Size, Data_Desc3+Seg_32,T1_LDT
Desc Stack_1,0,T1_Stack_Size,Data_Desc3+Seg_32,T1_LDT
Desc Stack0_1,0,T1_Stack0_Size,Data_Desc+Seg_32,T1_LDT
Desc Gate_1,Task_0,0,TaskGate_Desc3,T1_LDT
T1_LDT_Size Equ $-T1_LDT


; A T2 taszk lokális deszkriptorai

T2_LDT Label Qword
Desc Code_2, 0,T2_Code_Size, Code_Desc3+Seg_32,T2_LDT
Desc Data_2, 0,T2_Data_Size, Data_Desc3+Seg_32,T2_LDT
```

```
Desc Stack_2,0,T2_Stack_Size,Data_Desc3+Seg_32,T2_LDT
Desc Stack0_2,0,T2_Stack0_Size,Data_Desc+Seg_32,T2_LDT
Desc Gate_2,Task_0,0,TaskGate_Desc3,T2_LDT
T2_LDT_Size Equ $-T2_LDT

PM_Data Ends
```

Mivel az LDT-ben lévő deszkriptorok báziscíme is függ a program aktuális memóriapozíciójától, ezért szükség van egy eljárásra, amely az InitGDT-hez hasonlóan megfelelően módosítja a lokális deszkriptorokat. Alkalmazhatjuk erre a GDT-nél használt Seg2Base makrót, csak itt az ES:DI nem a GDT-re, hanem az adott lokális táblázatra mutat.

```
RM_Seg Segment
       Assume cs:RM_Seg, ds:RM_Seg, es:PM_Data

InitLDT Proc

    mov  ax,PM_Data             ; es = az LDT-k szegmense
    mov  es,ax

; A T1 taszk lokális deszkriptorai

    lea  di,PM_Data:T1_LDT      ; di = a T1 taszk LDT címe

    Seg2Base T1_Code            ; Code_1
    Seg2Base T1_Data            ; Data_1
    Seg2Base T1_Stack           ; Stack_1
    Seg2Base T1_Stack0          ; Stack0_1

; A T2 taszk lokális deszkriptorai

    lea  di,PM_Data:T2_LDT      ; di = a T2 taszk LDT címe

    Seg2Base T2_Code            ; Code_2
    Seg2Base T2_Data            ; Data_2
    Seg2Base T2_Stack           ; Stack_2
    Seg2Base T2_Stack0          ; Stack0_2

    ret

InitLDT Endp
```

Az LLDT utasításra itt nincs szükség, mivel a T0 csak GDT-ben lévő deszkriptorokat használ, a másik két taszknak pedig a TSS-ben állítjuk be az LDTR értékét.

A felügyelő szintű program (T0 taszk) az előzőekhez hasonlóan inicializálja a két 3. privilégiumszintű taszk TSS-ét az LDT-vel együtt, majd elkezd kapcsolgatni a két taszk között.

```
pm.inc

PM_Code Segment

PM_Main:
    lea   edx,TSS_1              ; DS:EDX = az 1. taszk TSS-e
    ...

    mov   [edx].TSS_LDTR,LDT_1

    lea   edx,TSS_2              ; DS:EDX = a 2. taszk TSS-e
    ...

    mov   [edx].TSS_LDTR,LDT_2


    mov   ecx,10000h
PM_1:
    Jump32_32 Task_1,0           ; ugrás az 1. taszkra
    Jump32_32 Task_2,0           ; ugrás a 2. taszkra

    loop PM_1

    ret

PM_Code Ends
```

A T2 taszk akár most is megpróbálhat sebezhető adatszegmenseket és TSS deszkriptorokat keresni, viszont az LDT-k és a taszkkapuk jóvoltából most nem fog sikerülni neki.

A lokális deszkriptorok segítségével tehát hatásos védelmet lehet kialakítani, és megakadályozható, hogy az azonos privilégiumszinten lévő taszkok elérjék egymás szegmenseit, és egyéb objektumokat.

2.7. Privilégiumszintek és I/O utasítások

Az eddigi 3. szintű taszkok csak egyszerűbb feladatokat végeztek el, egy számlálót növeltek meg, a képernyőre írtak, stb. A következő példákban a taszkok már I/O utasításokat is végrehajtanak, és azt nézzük majd meg, hogyan lehet ezeket a különböző privilégiumszinteken, illetve kapuk és I/O térképek segítségével szabályozni.

Vizsgáljuk meg először, hogy mi történt volna, ha például a T1 taszk végrehajtotta volna az in al,61h utasítást! A taszkok kezdeti EFLAGS értéke eddig 0 volt a TSS-ben, amely az első taszkváltás után azonnal 2-re változott, mivel az 1. bit mindig 1-re van állítva. Az EFLAGS-ban lévő IOPL bitek értéke tehát 0 volt, vagyis I/O műveleteket csak a 0. privilégiumszinten lehetett végrehajtani.

A 3. szinten végzett port olvasás viszont nem okoz azonnal általános védelmi hibát, hanem a processzor előbb megkeresi az adott porthoz tartozó bitet az I/O bittérképben. A TSS-ekben az TSS_IOBASE mezőt eddig nem állítottuk be, vagyis az értéke 0 volt. Ezt a processzor úgy értelmezi, hogy az I/O térkép a TSS báziscímétől kezdve helyezkedik el, és a mérete megegyezik a TSS limitjével (104 byte). A 61h port helyén, vagyis a 12. byte-on az 1. szintű veremmutató (ESP1) található, amelyet most nem használunk, ezért az értéke 0 volt. És mivel a processzor egy 0 bitet talál a megfelelő pozíción, végrehajtja az in al,61h utasítást, és hasonló módon engedélyezi több száz I/O port elérését is a felhasználói taszknak.

Ettől kezdve, ha nem használjuk a I/O térképet, akkor a TSS_IOBASE értékét a TSS határ fölé állítjuk be, és ajánlatos a többi nem használt mezőnek is érvénytelen értéket megadni.

Most a T1 taszk IOPL értékét 3-ra állítjuk, vagyis a felhasználói szinten nyugodtan használható I/O utasítás, a T2 taszknál viszont 1-re állítjuk, ezáltal csak a 0. és a (jelenleg nem használt) 1. szint végezhet I/O műveletet. Mivel a 61h port olvasása nincs hatással semmire, ezért az IN AL,61h utasítást próbáljuk majd ki a két taszkkal, a felhasználói és a felügyelői szinten (megszakításkapun keresztül) egyaránt, majd kiírjuk a képernyőre a portról olvasott értéket. A kivételkezelőt is módosítani fogjuk, hogy kezelje az esetleges általános védelmi hibát, és „emulálja" az I/O utasítást.

pm.inc

```
PM_Code Segment
        Assume cs: PM_Code, ds:PM_Data, es:PM_Data

PM_Main:

    lea  esi,Ex7_Str
    call WriteStr

    mov  ax,Task_0
    ltr  ax

; Az 1. taszk TSS-ének beállítása

    lea  edx,TSS_1

    mov  [edx].TSS_CS,Code_1+RPL3
    mov  [edx].TSS_EIP,0

    mov  ax,Data_1+RPL3
    mov  [edx].TSS_DS,ax
    mov  [edx].TSS_ES,ax
    mov  [edx].TSS_FS,ax
    mov  [edx].TSS_GS,ax

    mov  [edx].TSS_SS,Stack_1+RPL3
    mov  [edx].TSS_ESP,T1_Stack_Size
    mov  [edx].TSS_SS0,Stack0_1
    mov  [edx].TSS_ESP0,T1_Stack0_Size

    mov  [edx].TSS_EFLAGS,3002h    ; IOPL=3

    mov  [edx].TSS_IOBASE,-1       ; IOBASE > TSS határ

    mov  [edx].TSS_LDTR,LDT_1

; Az 2. taszk TSS-ének beállítása

    lea  edx,TSS_2

    mov  [edx].TSS_CS,Code_2+RPL3
    mov  [edx].TSS_EIP,0

    mov  ax,Data_2+RPL3
    mov  [edx].TSS_DS,ax
    mov  [edx].TSS_ES,ax
    mov  [edx].TSS_FS,ax
    mov  [edx].TSS_GS,ax

    mov  [edx].TSS_SS,Stack_2+RPL3
    mov  [edx].TSS_ESP,T2_Stack_Size
```

```
    mov   [edx].TSS_SS0,Stack0_2
    mov   [edx].TSS_ESP0,T2_Stack0_Size

    mov   [edx].TSS_EFLAGS,1002h    ; IOPL=1

    mov   [edx].TSS_IOBASE,-1       ; IOBASE > TSS határ

    mov   [edx].TSS_LDTR,LDT_2

; Ugrás a két taszkra.

    lea   esi,T1_Str
    call WriteStr
    Jump32_32 Task_1,0

    lea   esi,T2_Str
    call WriteStr
    Jump32_32 Task_2,0

    ret

PM_Code Ends


PM_Data Segment

Ex7_Str db '7. Példa: olvasása a 61h portról.',13,13
        db 'XX = közvetlen I/O olvasás',13
        db 'YY = INT 30h megszakításon keresztül',13
        db 'ff = a 13. kivételkezelõ „látszólagos" '
        db 'értéke',13

        db 13,'Olvasott értékek  : 0000XXYY',0
T1_Str  db 13,'1. Taszk, IOPL=3  : ',0
T2_Str  db 13,'2. Taszk, IOPL=1  : ',0

PM_Data Ends
```

A 0. privilégiumszintű tesztet a már meglévő INT 30h kiegészítésével végezzük el. Két új funkciót hozunk létre: a 10h-val a DX-ben megadott portról beolvasunk egy byte-ot, amit a DL-ben adunk vissza, a 11h-val pedig az AH-ban lévő byte-ot írjuk ki a DX portra.

Funkció	Sorszám	Paraméter
Olvasás I/O portról	AL = 10h	DX = port
Irás I/O portra	AL = 11h	DX = port, AH = érték

2.9. ábra Az INT 30h új funkciói

```
PM_Code Segment

Int_30 Proc

   push eax ds

   push Data32                ; DS = 0. szintû adatszegmens
   pop  ds

F_0:
   ...

F_10:
   cmp  al,10h                ; 16. funkció?
   jne  F_11

   in   al,dx                 ; olvasás az adott portról
   mov  dl,al                 ; a DL-be

   jmp  F_End

F_11:
   cmp  al,11h                ; 17. funkció?
   jne  F_End

   mov  al,ah                 ; írás az adott portra
   out  dx,al

F_End:
   pop  ds eax
   iretd

Int_30 Endp

PM_Code Ends
```

Az eddigi 13. kivételkezelő csak egyszerűen kiírta a regisztereket, és kilépett a védett módú programból. Ennél most többre van szükség, hogy a kivételt okozó tiltott I/O műveletek a taszkok számára észrevétlenek legyenek. A veremben lévő CS:EIP cím felhasználásával megállapítható a kivételt okozó utasítás. A régi CS:EIP értékét a DS:EAX regiszterekbe töltjük be, amely elég veszélyes, és csak akkor működik, ha az adott kódszegmens olvasható is, és a határral sincs baj. Ennél sokkal biztonságosabb lenne, ha megállapítanánk a CS szelektor által jelzett kódszegmens deszkriptornak a báziscímét, és ehhez hozzáadnánk az EIP régi értékét, majd a Flat32 segítségével vizsgálnánk meg a hibás utasítást.

Most csak az IN utasítás két formáját ellenőrizzük, és az IN AL,DX vagy az IN AL,port típusú 1 illetve 2 byte-os utasítások esetén az AL-be FFh byte-ot helyezünk el, és visszaadjuk a taszknak a vezérlést. Hasonló módon emulálható a portokkal kapcsolatos többi utasítás is (prefixek és egyebek figyelembe vételével).

idt.inc

```
Exc_13:
    push eax ds

    mov  eax,ss:[esp+16]      ; ax = CS
    mov  ds,ax
    mov  eax,ss:[esp+12]      ; eax = EIP

    mov  eax,ds:[eax]         ; eax = hibát kiváltó utasítás

    cmp  al,0E4h              ; in al,imm?
    je   Exc_13_in1
    cmp  al,0ECh              ; in al,dx?
    je   Exc_13_in2

    pop  ds eax

Exc_13_Abort:
    push offset PM_Data:Exc_Str13
    call Exc_Show             ; regiszterek kiírása
    jmp  Exc_Abort            ; kilépés a programból

Exc_13_in1:
    add  dword ptr ss:[esp+12],2
                              ; a hibás utasítás átugrása
    pop  ds eax
    add  esp,4                ; hibakód kiszedése a veremből
    mov  al,0ffh              ; a látszólagos érték
    iretd                     ; a taszk folytatása

Exc_13_in2:
    add  dword ptr ss:[esp+12],1
                              ; a hibás utasítás átugrása
    pop  ds eax
    add  esp,4                ; hibakód kiszedése a veremből
    mov  al,0ffh              ; látszólagos érték
    iretd                     ; a taszk folytatása
```

Az eltérő IOPL értékű taszkok tehát beolvasnak egy byte-ot a 3. és a 0. privilégiumszinten is, majd ezt kiírják a képernyőre. Az IOPL = 1 taszk esetén a 3. szintű olvasás eredménye FFh lesz, amely jelzi a kivételkezelő aktivizálódását.

task.inc

```
T1_Code Segment Para Public Use32
        Assume cs: T1_Code, ds: T1_Data
T1_Code_Start Label Byte

L1:
    mov  al,10h                 ; olvasás a 61h portról
    mov  dx,61h
    int  30h                    ; megszakításon keresztül

    in   al,61h                 ; olvasás közvetlenül
    mov  dh,al

    mov  al,1                   ; DL és DH kiírása
    int  30h

    Jump32_32 Gate_1,0          ; visszaugrás a hívó taszkra

    jmp  L1

T1_Code_Size Equ $-T1_Code_Start
T1_Code Ends
```

2.8. I/O műveletek bittérképpel

Az IOPL értékének beállításával egy taszkon belül az egyes privilégiumszintek számára az összes I/O művelet engedélyezhető, vagy letiltható. Legtöbbször viszont szükség van arra is, hogy az I/O portokhoz való hozzáférést csak részlegesen korlátozzuk.

Ez megvalósítható például a 13. kivételkezelőn keresztül is. Az IOPL 0-ra állításával az alsóbb szinteken minden I/O művelet kivételt generál, majd az adott port (a hibás utasításban lévő operandus, illetve a DX regiszter) vizsgálatával eldönthető, hogy az adott művelet végrehajtható-e, vagy sem. Ha az utasítás engedélyezett, akkor a 0. szinten futó kivételkezelő elvégezheti a kívánt műveletet, majd visszaadhatja a vezérlést a taszknak.

Az I/O bittérkép használatával viszont mindez sokkal hatékonyabban végezhető el, mivel nincsen szükség a kivételkezelő meghívására minden egyes I/O műveletnél. A processzor automatikusan dönti el a TSS-ben lévő térkép alapján, hogy engedélyezi az adott porthoz való hozzáférést, vagy általános kivételt generál.

A 13. kivételkezelőt kibővítjük, hogy ne csak a 8 bites port olvasását ismerje fel, hanem a 16 és 32 bites portok írását és olvasását is. Az IN esetén továbbra is FFh értéket adunk vissza. A hibás utasítás átugrásához szükség van az utasítások méretére is, amely 1, 2 és 3 byte-os lehet. Mivel a CLI és az STI utasítások is kivételt okoznak, ezért ezeket is célszerű kezelni; most egyszerűen csak figyelmen kívül hagyjuk őket, és az IF bitet változatlanul hagyjuk. Ha a 13. kivételt más utasítás okozta, akkor az eddigiekhez hasonlóan kiírjuk a regisztereket, visszakapcsolunk valós módba, és kilépünk a programból.

idt.inc

```
Exc_13:
    push eax ds

    mov  eax,ss:[esp+16]      ; ax = CS
    mov  ds,ax
    mov  eax,ss:[esp+12]      ; eax = EIP

    mov  eax,ds:[eax]         ; eax = hibát kiváltó utasítás

    cmp  al,0E4h              ; in al,imm?
    je   Exc_13_in2
    cmp  ax,066E5h            ; in ax,imm?
    je   Exc_13_in3
    cmp  al,0E5h              ; in eax,imm?
    je   Exc_13_in2

    cmp  al,0ECh              ; in al,dx?
    je   Exc_13_in1
    cmp  ax,066EDh            ; in ax,dx?
    je   Exc_13_in2
    cmp  al,0EDh              ; in eax,dx?
    je   Exc_13_in1

    cmp  al,0E4h              ; out imm,al?
    je   Exc_13_out2
    cmp  ax,066E5h            ; out imm,ax?
    je   Exc_13_out3
    cmp  al,0E5h              ; out imm,eax?
    je   Exc_13_out2

    cmp  al,0ECh              ; out dx,al?
    je   Exc_13_out1
    cmp  ax,066EDh            ; out dx,ax?
    je   Exc_13_out2
    cmp  al,0EDh              ; out dx,eax?
    je   Exc_13_out1
```

```
    cmp  al,0FAh                ; cli?
    je   Exc_13_cli
    cmp  al,0FBh                ; sti?
    je   Exc_13_sti

    pop  ds eax

Exc_13_Abort:
    push offset PM_Data:Exc_Str13
    call Exc_Show               ; regiszterek kiírása
    jmp  Exc_Abort              ; kilépés a programból


Exc_13_in3:                     ; 3 byte-os utasítás átugrása
    inc  dword ptr ss:[esp+12]
Exc_13_in2:                     ; 2 byte-os utasítás átugrása
    inc  dword ptr ss:[esp+12]
Exc_13_in1:                     ; 1 byte-os utasítás átugrása
    inc  dword ptr ss:[esp+12]
    pop  ds eax
    add  esp,4                  ; hibakód kiszedése a veremből
    mov  al,0ffh                ; látszólagos érték
    iretd                       ; a taszk folytatása

Exc_13_out3:                    ; 3 byte-os utasítás átugrása
    inc  dword ptr ss:[esp+12]
Exc_13_out2:                    ; 2 byte-os utasítás átugrása
    inc  dword ptr ss:[esp+12]
Exc_13_out1:                    ; 1 byte-os utasítás átugrása
    inc  dword ptr ss:[esp+12]
    pop  ds eax
    add  esp,4                  ; hibakód kiszedése a veremből
    iretd                       ; a taszk folytatása

Exc_13_cli:
Exc_13_sti:
    inc  dword ptr ss:[esp+12]
    pop  ds eax
    add  esp,4                  ; hibakód kiszedése a veremből
    iretd                       ; a taszk folytatása
```

A fentiekben nem szerepel az INS és az OUTS, és egyéb furfangos utasítások, mint például a REP CLI kezelése (a prefix miatt a kivételkezelő most nem ismeri fel a CLI-t).

A bittérképeket közvetlenül az egyes TSS-ek után helyezzük el, és a T1 taszknak engedélyezzük (bitek = 0) a 0-3ffh közötti portok elérését, a T2 taszknak pedig megtiltjuk (bitek = 1). Az első 400h port használata a leg-

gyakoribb, az e felettieket a TSS limit segítségével letiltjuk mindkét taszknál.

task.inc

```
PM_Data Segment

TSS_0 TSS_Struc <>              ; A 0. taszk TSS-e

TSS_1 TSS_Struc <>              ; Az 1. taszk TSS-e
IOBASE_1 = $-TSS_1              ; I/O térkép báziscíme
db 400h/8 dup (0)               ; bittérkép a 0-3ffh portokhoz
TSS_1_Size = $-TSS_1            ; T1 taszk TSS-ének hossza

TSS_2 TSS_Struc <>              ; A 2. taszk TSS-e
IOBASE_2 = $-TSS_2              ; I/O térkép báziscíme
db 400h/8 dup (0ffh)            ; bittérkép a 0-3ffh portokhoz
TSS_2_Size = $-TSS_2            ; 2. taszk TSS-ének hossza


PM_Data Ends
```

A TSS-ek méretét jelző 2 szimbólumra (TSS_1_Size, TSS_2_Size) azért van szükség, hogy a GDT-ben helyesen állítsuk be a deszkriptorok határát:

gdt.inc

```
Descriptor Task_0, 0,TSS_Size,      TSS_Desc
Descriptor Task_1, 0,TSS_1_Size,    TSS_Desc
Descriptor Task_2, 0,TSS_2_Size,    TSS_Desc
```

Ha például a T2 taszknak csak a 61h portot szeretnénk engedélyezni, és az összes többit letiltani, akkor az alábbi bittérképet kell definiálni.

```
TSS_2 TSS_Struc <>                  ; A 2. taszk TSS-e
IOBASE_2 = $-TSS_2                  ; I/O térkép báziscíme
db 12 dup (-1)                          ; 0-5fh portok letiltá-
sa
db 11111101b                            ; a 61h port engedélye-
zése
TSS_2_Size = $-TSS_2            ; a 68h feletti portok tiltása
```

A TSS-ben mindkét taszknál még be kell állítani a TSS_IOBASE-t a bittérképek kezdetére. Az IOPL értéke pedig 1 lesz, így tehát a felhasználói szinten kivétel keletkezik a CLI és az STI hatására, illetve a port I/O utasítások esetén, ha a térkép megfelelő bitje 1.

pm.inc

```
PM_Code Segment
        Assume cs: PM_Code, ds:PM_Data, es:PM_Data
PM_Main:

    lea  esi,Ex8_Str
    call WriteStr

    lea  edx,TSS_1              ; DS:EDX = az 1. taszk TSS-e

    ...

    mov  [edx].TSS_EFLAGS,1002h          ; IOPL=1
    mov  [edx].TSS_IOBASE,IOBASE_1       ; Az I/O térkép
                                         ; báziscíme
    lea  edx,TSS_2              ; DS:EDX = a 2. taszk TSS-e

    ...

    mov  [edx].TSS_EFLAGS,1002h          ; IOPL=1
    mov  [edx].TSS_IOBASE,IOBASE_2       ; Az I/O térkép
                                         ; báziscíme

    lea  esi,T1_Str
    call WriteStr
    Jump32_32 Task_1,0          ; ugrás az 1. taszkra

    lea  esi,T2_Str
    call WriteStr
    Jump32_32 Task_2,0          ; ugrás a 2. taszkra

    ret

PM_Code Ends

PM_Data Segment

Ex8_Str db '8. Példa: byte olvasása a 61h portról '
        db 'I/O bittérképpel.',13,13
        db 'XX = közvetlen I/O olvasás',13
        db 'YY = INT 30h megszakításon keresztül',13
        db 'ff = a 13. kivételkezelõ „látszólagos" '
        db 'értéke',13

        db 13,'Olvasott értékek: 0000XXYY',0
T1_Str  db 13,'1. Taszk, IOPL=1, 61h port bitje 0: ',0
T2_Str  db 13,'2. Taszk, IOPL=1, 61h port bitje 1: ',0

PM_Data Ends
```

A T2 taszk tehát most sem tud a 61h portról olvasni a 3. szinten (a bittérkép miatt), de az IF bitet sem tudja már megváltoztatni. Akár az OUT 61h,EAX utasítást is végrehajthatja (látszólag), viszont ezek a hibák már nincsenek hatással a rendszer egészére.

task.inc

```
T2_Code Segment Para Public Use32
        Assume cs: T2_Code, ds: T2_Data
T2_Code_Start Label Byte

L2:
    cli

    mov  al,10h                 ; olvasás a 61h portról
    mov  dx,61h
    int  30h                    ;  megszakításon keresztül

    in   al,61h                 ; olvasás közvetlenül
    mov  dh,al

    mov  al,1                   ; DL és DH kiírása
    int  30h

    sti

    Jump32_32 Gate_2,0          ; visszaugrás a hívó taszkra

    jmp  L2

T2_Code Size Equ $-T2_Code_Start
T2_Code Ends
```

2.9. Illeszkedő kódszegmensek és call-kapuk

Az eddigiek során olvasható és végrehajtható kódszegmenseket használtunk, és a kapuk közül csak a megszakításkapukat és a taszkkapukat alkalmaztunk. Ebben a példában az illeszkedő kódszegmenseket és a callkapukat is kipróbáljuk, és megnézzük, hogyan befolyásolják ezek a privilégiumszinteket.

A privilégiumszintek változását az előzőkhöz hasonlóan I/O utasításokkal és az IOPL bitekkel teszteljük. Két eljárást kell tehát létrehozni: egyet az illeszkedő kódszegmensben, egyet pedig a call-kapu számára.

Az illeszkedő IO_Test eljárás feladata egyszerű, mindössze beolvas egy byte-ot a 61h portól, majd RETF utasítással visszatér (FAR CALL-al hívjuk meg). Az illeszkedő kódszegmensek hívása esetén az aktuális privilégiumszint (CPL) nem változik, és az IN működését továbbra is a taszk IOPL szintje határozza meg.

```
IO_Test Proc

    in  al,61h
    retf

IO_Test Endp
```

A call-kapun keresztül hívott eljárás (IO_Test2) privilégiumszintje a kapuban lévő kódszegmens szelektor RPL mezőjével lesz egyenlő, hiszen ez a szelektor töltődik be a CS-be, és így az RPL egyben a CPL is lesz. A kapu DPL mezője pedig azt határozza meg, hogy mely szintekről hívható meg. A kapu lokális deszkriptortáblázatban való elhelyezésével csak bizonyos taszkok számára lehet elérhetővé tenni az adott eljárást.

A call-kapuk lehetőségeinek illusztrálására az IO_Test2 eljárásban két funkciót is létrehozunk, a portról olvasást, és a portra írást, továbbá a paramétereket nem regiszterekben, hanem a vermen keresztül adjuk át.

A call-kapuk egyik nagy előnye, hogy a felhasználói veremben elhelyezett paramétereket a processzor automatikusan átmásolja a magasabb szintű verembe. Ez azért hasznos, mivel a magasabb privilégiumszintre való kapcsolással együtt a verem is megváltozik, ezért a felhasználói veremben lévő adatokat a magasabb szintek csak nehezen érhetnék el.

	-20	1. paraméter
	-16	2. paraméter
	-12	3. paraméter
	-8	a hívó CS
	-4	a hívó EIP
a jelenlegi SS:EBP ⇨	0	az EBP eredeti tartalma
	+4	az EAX eredeti tartalma
a jelenlegi SS:ESP ⇨	+8	az EDX eredeti tartalma

2.10. ábra A verem tartalma az IO_Test2 eljárásban

Az IO_Test2 eljárásnak 3 darab 32 bites paramétert adunk át: az első jelzi a funkciót (0 = olvasás, 1 = írás), a második a kívánt I/O portot, a harmadik pedig a kiírandó értéket. A paramétereket az EBP regiszterrel érhetjük majd el, a verem állapota (az EBP mutatóhoz képest) az alábbi ábrán szerepel.

pm.inc

```
IO_Seg Segment Para Public Use32
IO_Seg_Start Label Byte
      Assume cs: IO_Seg, ds:Nothing, es:Nothing


IO_Test2 Proc

    push ebp
    mov  ebp,esp
    push eax
    push edx

    mov  eax,[ebp+20]          ; EAX = funkció sorszáma

    cmp  eax,0                 ; I/O olvasás?
    je   IO_Read
    cmp  eax,1                 ; I/O írás?
    je   IO_Write
    jmp  IO_End                ; érvénytelen funkció

IO_Read:
    mov  edx,[ebp+16]          ; EDX = port sorszáma
    mov  eax,[ebp-4]           ; EAX = az eredeti EAX

    in   al,dx                 ; olvasás a megadott portról

    mov  [ebp-4],eax           ; az új EAX tárolása
    jmp  IO_End

IO_Write:
    mov  edx,[ebp+16]          ; EDX = port sorszáma
    mov  eax,[ebp+12]          ; EAX = a kiírandó érték

    out  dx,al                 ; a byte kiírása

IO_End:
    pop  edx
    pop  eax
    pop  ebp

    retf 12                    ; a 3 paraméter kiszedése
                               ; a veremből
```

```
IO_Test2 Endp

IO_Seg_Size Equ $-IO_Seg_Start
IO_Seg Ends
```

A 0. funkció esetén a beolvasott értéket az EAX regiszter veremben tárolt tükörképében helyezzük el, mivel azt adjuk majd vissza a hívónak. Az eljárás végén a 0. szintű veremből még el kell távolítani a 3 duplaszót is (RETF 12), a hívó szint veméből viszont a processzor automatikusan törli a paramétereket. Bár ezzel a két funkcióval csak a 8 bites I/O műveleteket lehet elvégezni, de az eljárás könnyen kiegészíthető, hogy a 16 és 32 bites operandusok, illetve a string I/O utasítások is elérhetők legyenek a taszkok számára.

Létre kell még hozni a szükséges deszkriptorokat is a GDT-ben (itt mindkét taszk láthatja). A call-kapu esetén viszont nem használhatjuk a Descriptor makrót, mivel a kapuknak kicsit eltérő formátuma van. Az alábbi Call_Gate3 makró egy 3. szintű call-kaput definiál, a paraméterei pedig a szelektor neve, a belépési pont (CS:EIP), valamint az átmásolandó duplaszavak száma (0–63).

defs.inc

```
CallGate_Desc3 Equ Present+DPL3+Sys_Desc+Sys_CallGate

Call_Gate3 Macro Name,Sel,Off,Param
    Name Equ $-GDT
    dw   small Off
    dw   Sel
    db   Param
    db   CallGate_Desc3
    dw   0
Endm
```

A deszkriptorok privilégiumszintjét (DPL) az illeszkedő kódszegmens (IOCode) és a kapu kódszegmens (IOCode2) esetén 0-ra állítjuk. Az IOCode tehát meghívható bármely privilégiumszintről, viszont a CPL változatlan marad, az IOCode2 kódszegmenset pedig csak a call-kapun (IOGate) keresztül hívhatják meg a 3. szintű taszkok.

gdt.inc

```
Descriptor IOCode,0,IO_Seg_Size,\
          Present+DPL0+Seg_Desc+Seg_32+Code_ERC
Descriptor IOCode2,0,IO_Seg_Size,\
          Present+DPL0+Seg_Desc+Seg_32+Code_ER
Call_Gate3 IOGate,IOCode2,IO_Test2,3
```

Ha az IOCode kódszegmens deszkriptorában a DPL-t például 7-re állítanánk, akkor azzal megtilthatnánk a felügyelői szintnek, hogy meghívja az IO_Test eljárást.

Az I/O műveletek különböző módon történő tesztelése van már csak hátra. Az IO_Test eljárás eredménye csak akkor lesz helyes, ha az IOPL = 3, egyébként 0ffh-t kapunk, az IOGate kapun keresztül meghívott IO_Test2 pedig mindig helyes értéket ad.

task.inc

```
T1_Code Segment Para Public Use32
        Assume cs: T1_Code, ds: T1_Data
T1_Code_Start Label Byte

L1:
    CallFar IOCode,IO_Test    ; az illeszkedõ
                              ; kódszegmens hívása
    mov  dh,al                ; az eredmény tárolása

    push 0                    ; 0. funkció, I/O olvasás
    push 61h                  ; a port sorszáma
    push 0                    ; az olvasáshoz felesleges.
    CallFar IOGate,0          ; call-kapu hívása
    mov  dl,al                ; az eredmény tárolása

    mov  al,1                 ; DL és DH kiírása
    int  30h

    Jump32_32 Gate_1,0        ; visszaugrás a hívó taszkra

    jmp  L1

T1_Code_Size Equ $-T1_Code_Start
T1_Code Ends
```

3. Lapozás és a virtuális üzemmód

3.1. A lapozás engedélyezése, a táblázatok beállítása

A következő néhány példában a lapozásról lesz szó. Legelőször a lapozás engedélyezésével az alsó 16 MB memóriát leképezzük „saját magára", vagyis ezen a területen a lineáris címek megegyeznek majd a fizikai címekkel. A szöveges képernyőt pedig a 0b8000h mellett elérhetővé tesszük az 1 GB (40000000h) lineáris címen is.

Ehhez először szükség van egy 4 KB méretű lapcímtárra, amelynek a címét a CR3 regiszterben helyezzük el. Mivel minden egyes 4 MB-os területhez a lapcímtárban egy 32 bites bejegyzés tartozik, ezért a 16 MB leképezéséhez 4 darab második szintű laptáblázatot is létre kell hozni. Ezeknek a laptáblázatoknak a címét a lapcímtár első négy bejegyzésében kell elhelyezni. A képernyő leképezéséhez pedig a lapcímtár 256. elemében (256*4MB = = 1 GB) kell egy laptáblázatot kialakítani, amelyben az első duplaszóban a báziscímet 0b8000h-ra kell állítani. Az érvényes bejegyzésekben persze a P (Present) bitet 1-re kell állítani, illetve igény szerint a R/W és a U/S biteket is.

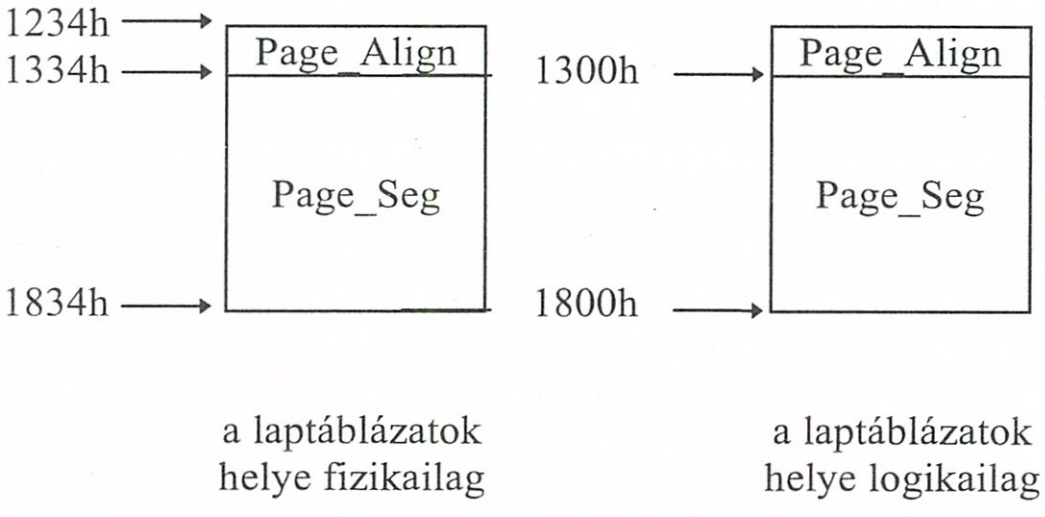
Általában a laptáblázatok elhelyezésekor a legnagyobb gondot az okozza, hogy ezek 4 KB-hoz igazodjanak. Ez azért nehézkes, mert erre az igazításra sem statikusan (fordítás+linkelés), sem dinamikusan (egyszerű memóriafoglalással) nincsen lehetőség. A legkönnyebb megoldás, ha kiszemelünk egy abszolút címet (pl. 90000h, vagy 2 MB), és ott alakítjuk ki a lapozáshoz szükséges táblázatokat, felülírva a memória eredeti tartalmát. Mivel ez elég primitív módszer, ezért a 4 KB igazítást inkább másképp végezzük el.

A leképezéshez tehát 5 darab 4 KB méretű táblázatra van szükség: 1 lapcímtárra és 5 laptáblázatra. Ezeket statikusan, egy külön szegmensben (Page_Seg) helyezzük el, mégpedig úgy, hogy ne foglaljanak helyet a végrehajtható file-ban. A táblázatokat tartalmazó szegmens elé egy 4 KB-os szegmenset (Page_Align) rakunk, és ezt használjuk fel az igazításhoz.

page.inc

```
Page_Align Segment Para Public

db 4096 dup (?)

Page_Align Ends


Page_Seg Segment Para Public Use32

PageDir                 dd 1024 dup (?)      ; lapcímtár

PageTable0              dd 1024 dup (?)      ; 0-4 Mb
PageTable1              dd 1024 dup (?)      ; 4-8 Mb
PageTable2              dd 1024 dup (?)      ; 8-12 Mb
PageTable3              dd 1024 dup (?)      ; 12-16 Mb
PageTableV              dd 1024 dup (?)      ; 0b8000h -> 1 Gb

Page_Seg Ends
```

A program futása közben már tudjuk, hogy az egyes szegmensek pontosan hol helyezkednek el a memóriában, tehát megállapítható az is, hogy a Page_Seg milyen memóriacímen kezdődik. Ezután már csak el kell tolni – látszólag – a szegmenset annyival, hogy az alsó 12 bit 0 legyen, vagyis hogy 4 KB-hoz igazodjon. Ekkor a Page_Seg kezdete belelóg a Page_Align szegmensbe, amelyet éppen ezért hoztunk létre. Az egyes táblázatok tényleges címét tehát úgy kapjuk meg, hogy a Page_Seg látszólagos báziscíméhez hozzáadjuk a szegmensen belüli ofszetcímet.



3.1. ábra A laptáblázatok áthelyezése az 1234:0000 címen

Mivel minden egyes hivatkozás esetén szükség van erre a látszólagos báziscímre, ezért a táblázatok kialakítása közben az EBX regiszter végig ezt az értéket fogja tartalmazni. A védett módba kapcsolás után az EnablePaging eljárás meghívásával engedélyezzük a lapozást. Az eljárás az EBX-ben elhelyezi a kiigazított báziscímet, inicializálja a táblázatokat, és a CR3 regisztert, majd a CR0 regiszter legfelső bitjét 1-re állítja. A valós módba kapcsolás előtt pedig a DissablePaging eljárást hívjuk meg, amely egyszerűen csak törli a PG bitet.

```
PM_Code Segment
        Assume cs:PM_Code, ds:Nothing, es:Nothing

EnablePaging Proc

    xor   ebx,ebx               ; igazítás 4K-ra
    mov   bx,Page_Align
    shr   ebx,8
    inc   ebx
    shl   ebx,12                ; ebx=a táblázatok címe

    call  InitPages             ; laptáblázatok beállítása

    mov   eax,ebx
    add   eax,offset PageDir
    mov   CR3,eax               ; lapcímtár báziscímének
                                ; beállítása

    mov   eax,CR0               ; lapozás engedélyezése
    or    eax,80000000h         ; PG bit = 1
    mov   CR0,eax

    ret

EnablePaging Endp

DissablePaging Proc

    mov   eax,CR0               ; lapozás letiltása
    and   eax,7fffffffh         ; PG = 0
    mov   CR0,eax
    ret

DissablePaging Endp

PM_Code Ends
```

Amikor engedélyezzük vagy letiltjuk a lapozást, akkor a kódszegmens adott pozíciójában a lineáris címnek mindig meg kell egyeznie a fizikai

címmel, különben a PG bit változtatása után teljesen máshol folytatódik az utasítások végrehajtása.

Az alábbi InitPages eljárás feladata, hogy a lapozás engedélyezése előtt a megfelelő módon beállítsa az egyes táblázatokat. Először a 4 laptáblázatot (PageTable0-PageTable3) kell feltölteni az egyes lapok címével, ehhez mindössze az egymás után következő dupla szavakban mindig 4096-al nagyobb értéket kell tárolni. A Page_Seg 32 bites látszólagos báziscíme az EBX-ben van.

```
InitPages Proc

    push es

    mov  ax,Flat32              ; ES = lineáris 0
    mov  es,ax

; Az alsó 16 Mb lineáris memória leképezése
; a fizikai memóriára.

    lea  edi,PageTable0; ES:EDI = az elsõ táblázat
    add  edi,ebx                ; valódi 4K-s lapra igazítás
    mov  eax,7                  ; jelenlévõ, írható,
                                ; felhasználói lapok.
    mov  ecx,4*1024             ; 4 teljes táblázat

Init_16M:
    stosd
    add  eax,4096               ; következõ lap ofszetcíme
    loop Init_16M

; A 0b8000h-n lévõ 4K leképezése az 1 Gb területre.

    lea  edi,PageTableV; ES:EDI = a videomemória
                                ; laptáblázata
    add  edi,ebx                ; kiigazítás
    mov  eax,0b8000h+7
    stosd                       ; egy 4K-s lap bejegyzése

; A többi bejegyzés érvénytelenítése

    xor  eax,eax                ; 0. bit: nincs jelen.
    mov  ecx,1024-1             ; 1 lap már megvan.
    rep  stosd

; A Lapcímtár beállítása

    lea  edi,PageDir            ; A teljes lapcímtár
```

```
    add  edi,ebx
    mov  ecx,1024
    xor  eax,eax
    rep  stosd                  ;  érvénytelenítése


    lea  edi,PageDir            ; ES:EDI = lapcímtár
    add  edi,ebx

    lea  eax,PageTable0; 0. bejegyzés
    add  eax,ebx
    add  eax,7
    stosd
    lea  eax,PageTable1; 1. bejegyzés
    add  eax,ebx
    add  eax,7
    stosd
    lea  eax,PageTable2; 2. bejegyzés
    add  eax,ebx
    add  eax,7
    stosd
    lea  eax,PageTable3; 3. bejegyzés
    add  eax,ebx
    add  eax,7
    stosd

    lea  edi,PageDir            ; ES:EDI = lapcímtár
    add  edi,ebx
    add  edi,256*4
    lea  eax,PageTableV; 256. bejegyzés (1 Gb)
    add  eax,7
    add  eax,ebx
    stosd

    pop  es
    ret

InitPages Endp
```

A kialakított laptáblázatokat kipróbálhatjuk, ha az EnablePaging eljárás meghívása után kiírunk két fehér pontot a képernyőre: egyet az igazi képernyő-memóriába (0b8000h), és egyet a virtuálisba (1 GB). A két koordinátát (XY1 és XY2) az adatszegmensben helyezzük el azért, hogy az alsó 16 MB lapjait is kipróbáljuk.

pm.inc

```
PM_Code Segment
        Assume cs: PM_Code, ds:PM_Data, es:PM_Data
PM_Main:
```

```
    lea  esi,Ex1_Str
    call WriteStr

    call EnablePaging          ; a lapozás engedélyezése

    mov  eax,0b8000h
    add  eax,XY1
    mov  word ptr fs:[eax],'.'+15*256

    mov  eax,1024*1024*1024    ; 1 Gb
    add  eax,XY2
    mov  word ptr fs:[eax],'.'+15*256

    call DissablePaging        ; a lapozás letiltása

    ret

PM_Code Ends


PM_Data Segment

Ex1_Str db '1. Példa: Lapozás.',13,13
   db 'A videomemória elérése a B8000h címen: [ ]',13
   db 'A videomemória elérése az 1 Gb címen:  [ ]'
   db 13,13,0

XY1 dd 5*160+40*2              ; 5. sor közepe
XY2 dd 6*160+40*2              ; 6. sor közepe

PM_Data Ends
```

3.2. Egy taszk elhelyezkedése az 1, 2 és 3 GB-os területen

Ebben a példában játszadozunk egy kicsit a gigabyte-okkal. Létrehozunk egy taszkot, amely látja a teljes 4 GB-os lineáris memóriát, vagyis a szegmensregiszterekben 0 báziscímű és 4 GB méretű szegmensek szelektorait helyezünk el. A taszk kódszegmense az 1 GB-on, az adatszegmense a 2 GB-on, a veremszegmense pedig a 3 GB-on lesz. Mindehhez persze fizikailag még 1 MB kiterjesztett memória sem szükséges, a virtuális címeket a laptáblázatok segítségével alakítjuk ki.

Először hozzuk létre a lokális deszkriptortáblázatban a 4 GB méretű szegmensdeszkriptorokat és a szokásos taszkkaput. Ahhoz, hogy elérjük a

maximális szegmensméretet, a 20 bites határt FFFFFh-ra, a G bitet pedig 1-re állítjuk (Seg_Big szimbólum). Valójában a veremszegmens számára nincs szükség külön deszkriptorra, hiszen minden tulajdonsága megegyezik az adatszegmens deszkriporával, és ezért a Data_1 szelektort használjuk majd a verem címzéséhez is. Mivel a deszkriptorok báziscíme nem függ a program aktuális helyétől, ezért ezek módosítását sem kell elvégezni.

ldt.inc

```
PM_Data Segment

; A T1 taszk lokális deszkriptorai

T1_LDT Label Qword
Desc Code_1,0,0fffffh,Code_Desc3+Seg_32+Seg_Big,T1_LDT
Desc Data_1,0,0fffffh,Data_Desc3+Seg_32+Seg_Big,T1_LDT
Desc Gate_1,Task_0,0,TaskGate_Desc3,T1_LDT
T1_LDT_Size Equ $-T1_LDT

PM_Data Ends

; A lokális deszkriptortáblázat beállítása

RM_Seg Segment
      Assume cs:RM_Seg, ds:RM_Seg

InitLDT Proc

   ret

InitLDT Endp


RM_Seg Ends
```

A GDT a T0 taszk szegmensdeszkriptorain kívül a két TSS deszkriptort, illetve a T1 taszk LDT deszkriptorát (LDT_1) tartalmazza, valamint itt helyezzük el a T1 taszk 0. szintű vermét is (Stack0_1).

gdt.inc

```
GDT_Seg Segment

GDT Label Qword

   ...

Descriptor Task_0,0,TSS_Size,TSS_Desc
```

```
Descriptor Task_1,0,TSS_Size,TSS_Desc

Descriptor LDT_1,0,T1_LDT_Size,LDT_Desc

Descriptor Stack0_1,0,T1_Stack0_Size,Data_Desc+Seg_32

GDT_Len     Equ $-GDT           ; a GDT hossza

GDT_Seg Ends
```

A deszkriptorok után következhetnek a laptáblázatok. A T0 és a T1 taszkok számára két külön laptáblázatot hozunk létre, és a laptáblázatok báziscímeit a TSS-ek CR3 mezőjében helyezzük el. A felügyelői taszk továbbra is elérheti az alsó 16 MB memóriát (függetlenül attól, hogy fizikailag van-e ennyi a gépben), illetve a képernyőt az 1 GB címen.

A T1 taszk lapcímtárában összesen négy laptáblázatot kell kialakítani, egyet a kód számára (256. bejegyzés, 1 GB), egyet az adatok számára (512. bejegyzés, 2 GB), és egyet a verem számára (768. bejegyzés, 3 GB). Ezzel a három táblázattal a taszk csak a saját memóriaterületeit érheti el. Viszont ha a taszk meghívja az INT 30h megszakítást, illetve ha kivétel vagy egyéb keletkezik, akkor a 0. privilégiumszint kód-, adat-, és veremszegmenseire is szükség van. Ha ezek nincsenek definiálva az adott taszk laptáblázataiban, akkor az egész rendszer halálra van ítélve. (Laphiba keletkezne, de a kivételkezelő lapjai sem érhetőek el.)

Ha egyszerűen leképeznénk a 0. szint lapjait is a T1 laptáblázataiban, akkor azokat a taszk könnyedén felülírhatná, hiszen eléri a teljes lineáris memóriát. Ehelyett inkább a lapcímtár 0. bejegyzésében olyan laptáblázat címét adjuk meg, amely az alsó 4 MB lapjait felügyelői szintűnek (U/S = 0) definiálja. Ekkor az alkalmazói szintről nem érhető el ez a terület (laphibát okoz), viszont a felsőbb szintekről már igen (ha CPL<3).

A két lapcímtár és a laptáblázatok a Page_Seg szegmensben lesznek, és a Page_Align szegmenset használjuk fel továbbra is a 4 KB-os igazításhoz.

page.inc

```
Page_Align Segment Para Public

db 4096 dup (?)                 ; a 4K illeszkedéshez

Page_Align Ends
```

```
Page_Seg Segment Para Public Use32

PageDir             dd 1024 dup (?)     ; T0 lapcímtár

PageTable0          dd 1024 dup (?)     ; 0-4 Mb
PageTable1          dd 1024 dup (?)     ; 4-8 Mb
PageTable2          dd 1024 dup (?)     ; 8-12 Mb
PageTable3          dd 1024 dup (?)     ; 12-16 Mb
PageTableV          dd 1024 dup (?)     ; 0b8000h -> 1 Gb


PageD1              dd 1024 dup (?)     ; T1 lapcímtár

PageT1C             dd 1024 dup (?)     ; 1 Gb, kódszegmens
PageT1D             dd 1024 dup (?)     ; 2 Gb, adatszegmens
PageT1S             dd 1024 dup (?)     ; 3 Gb, veremszegmens
PageT1M             dd 1024 dup (?)     ; 0-4Mb, a 0. szint
                                        ; számára

T1_Copy             db 3*4096 dup (?)   ; T1 szegmenseinek
                                        ; másolata

Page_Seg Ends
```

A 3 lap méretű T1_Copy azt a célt szolgálja, hogy a kód-, az adat-, és a veremszegmens ne találkozzon egy lapon. Például jelenleg a T1 kódszegmense nem tölt ki egy teljes lapot, ezért az adatok is átcsúsznának erre a lapra. Hogy ezt elkerüljük, minden szegmenset külön lapra másolunk, és az új címeket helyezzük el a laptáblázatok bejegyzéseiben.

A T1 taszk laptáblázatainak beállítása tehát három lépésből áll: először a szegmenseket saját lapra másoljuk, majd a második szintű táblázatokban megadjuk a felhasználói (1 GB, 2GB, 3 GB) és a felügyelői (0–4 MB) szintekhez tartozó lapok címét, végül a lapcímtár 0., 256., 512., és 768. bejegyzéseiben elhelyezzük a laptáblázatok báziscímét.

```
PM_Code Segment
        Assume cs:PM_Code, ds:Nothing, es:Nothing


InitPages_T1 Proc

    push ds es

    mov  ax,Flat32              ; ES = lineáris 0
    mov  es,ax
    mov  ds,ax
```

```
; A T1 taszk szegmenseinek átmásolása
; egy 4K-ra illeszkedõ területre

    lea   edi,T1_Copy          ; EDI = T1 másolat 0. lapja
    add   edi,ebx
    xor   esi,esi
    mov   si,T1_Code           ; ESI = T1 kódszegmensének
    shl   esi,4                ; lineáris címe
    mov   ecx,T1_Code_Size ; a szegmens hossza
    rep   movsb                ; átmásolás...

    lea   edi,T1_Copy+4096 ; EDI = T1 másolat 1. lapja
    add   edi,ebx
    xor   esi,esi
    mov   si,T1_Data           ; ESI = T1 adatszegmensének
    shl   esi,4                ; lineáris címe
    mov   ecx,T1_Data_Size ; a szegmens hossza
    rep   movsb                ; átmásolás...

    lea   edi,T1_Copy+8192 ; EDI = T1 másolat 2. lapja
    add   edi,ebx
    xor   esi,esi
    mov   si,T1_Stack          ; ESI = T1 veremszegmensének
    shl   esi,4                ;  lineáris címe
    mov   ecx,T1_Stack_Size    ; a szegmens hossza
    rep   movsb                ; átmásolás...

; A 3 szegmenshez tartozó laptáblázatok beállítása

    lea   edi,PageT1C          ; ES:EDI = a kódszegmens
                               ; táblázata
    add   edi,ebx              ; lineáris cím
    mov   eax,7                ; jelenlévõ, írható
                               ; és felhasználói lap
    add   eax,offset T1_Copy
    add   eax,ebx              ; EAX = T1 kódszegmensének
                               ; lineáris címe
    mov   es:[edi],eax

    add   edi,4096             ; ES:EDI = az adatszegmens
                               ; táblázata
    add   eax,4096             ; a következõ lap ofszetcíme
    mov   es:[edi],eax

    add   edi,4096             ; ES:EDI = a veremszegmens
                               ; táblázata
    add   eax,4096             ; a következõ lap ofszetcíme
    mov   es:[edi],eax

; A 0-4 MB lineáris memória leképezése a fizikai
; memóriára, a felügyelõi szint számára
```

```
    lea   edi,PageT1M          ; ES:EDI = elsõ 4 MB táblázata
    add   edi,ebx              ; valódi 4K-s lapra igazítás
    mov   eax,3                ; jelenlévõ, írható
                               ; és felügyelõi lapok.
    mov   ecx,1024             ; 1 táblázat (4M)
Init_T1:
    stosd
    add   eax,4096             ; a következõ lap ofszetcíme
    loop  Init_T1

; A Lapcímtár beállítása

    lea   edi,PageD1           ; A teljes lapcímtár
    add   edi,ebx
    mov   ecx,1024
    xor   eax,eax
    rep   stosd                ; érvénytelenítése


    lea   edi,PageD1           ; ES:EDI = lapcímtár
    add   edi,ebx

    lea   eax,PageT1M          ; 0. bejegyzés
    add   eax,ebx
    add   eax,3
    mov   [edi],eax

    lea   eax,PageT1C          ; 256. bejegyzés
    add   eax,ebx
    add   eax,7
    mov   [edi+256*4],eax

    lea   eax,PageT1D          ; 512. bejegyzés
    add   eax,ebx
    add   eax,7
    mov   [edi+512*4],eax

    lea   eax,PageT1S          ; 768. bejegyzés
    add   eax,ebx
    add   eax,7
    mov   [edi+768*4],eax

    pop   es ds
    ret

InitPages_T1 Endp

PM_Code Ends
```

A kódszegmenshez tartozó lapot definiálhattuk volna írásvédettnek is, mivel így az adatszegmens szelektorának segítségével a T1 taszk módosíthatja a saját kódját. Ha az 1 GB-on lévő lap csak olvasható, akkor minden írási kísérlet laphiba kivételt okoz.

Mivel a T1 taszk összes szegmensregisztere a 0 lineáris címre mutat, ezért az EIP és az ESP regisztereket a szegmensek helyének megfelelően kell beállítani. Az EIP-ben a kódszegmens báziscímét (40000000h, 1 GB), az ESP-ben pedig a veremszegmens tetejének címét (C0001000h, 3 GB + 4 KB) helyezzük el. A CS-be a kódszegmens deszkriptorának szelektorát töltjük be, az összes többi szegmensregiszterbe pedig az adatszegmens szelektorát (az RPL értéke természetesen mindenhol 3).

A CR3 regiszter beállításánál vigyázni kell, mivel ennek tartalma – eltérően például az általános regiszterektől – nem frissítődik minden taszkváltásnál. Vagyis nem elég, ha a felügyelői taszknál feltöltjük a CR3 regisztert a lapcímtár címével, mivel a következő alkalommal, amikor a T0 taszkra kapcsolunk, a CR3-ba 0 kerül (a TSS-ből), ami a gép elkerülhetetlen lefagyását eredményezi. Tehát a T0 taszk TSS-ében is meg kell adni a lapcímtár báziscímét (Page_Dir), és nem csak a T1 taszk TSS-ében (Page_D1).

pm.inc

```
PM_Code Segment
        Assume cs: PM_Code, ds:PM_Data, es:PM_Data
PM_Main:

    call EnablePaging

    lea  esi,Ex2_Str
    call WriteStr

    mov  dl,15
    call SetColor

    mov  ax,Task_0              ; az elsõ taszk kiválasztása
    ltr  ax

    lea  edx,TSS_0              ; DS:EDX=a T0 taszk TSS-e
    mov  eax,CR3
    mov  [edx].TSS_CR3,eax      ; lapcímtár tárolása


    lea  edx,TSS_1              ; DS:EDX = az 1. taszk TSS-e
```

```
    mov   [edx].TSS_CS,Code_1+RPL3      ; bázis = 0
    mov   [edx].TSS_EIP,40000000h       ; EIP=1 Gb

    mov   ax,Data_1+RPL3                ; bázis = 0
    mov   [edx].TSS_DS,ax
    mov   [edx].TSS_ES,ax
    mov   [edx].TSS_FS,ax
    mov   [edx].TSS_GS,ax

    mov   [edx].TSS_SS,ax
    mov   [edx].TSS_ESP,0c0000000h+T1_Stack_Size; 3 Gb

    mov   [edx].TSS_SS0,Stack0_1
    mov   [edx].TSS_ESP0,T1_Stack0_Size

    mov   [edx].TSS_EFLAGS,0

    mov   [edx].TSS_LDTR,LDT_1

    mov   eax,CR3
    add   eax,offset PageD1    ; EAX=T1 taszk lapcímtára
    mov   [edx].TSS_CR3,eax

    mov   ecx,10000h
PM_1:
    Jump32_32 Task_1,0         ; ugrás az 1. taszkra

    loop PM_1

    call DissablePaging        ; lapozás letiltása

    ret

PM_Code Ends



PM_Data  Segment

Ex2_Str db '2. Példa: A kódszegmens az 1 Gb-os, '
        db 'az adatszegmens a 2 Gb-os területen.'
        db 13,13
        db '1. Taszk számlálója:  [          ]'
        db ,13,13,0

PM_Data Ends
```

A T0 taszk tehát a lapozás engedélyezése után beállítja a TSS-eket, majd a T1 taszkot meghívja néhányszor, és végül a lapozás letiltása után kilép. A T1 taszk feladata a megszokott számláló növelése és képernyőre írá-

sa, mivel így ellenőrizhetjük az adatszegmenset, illetve a felügyelői szinthez tartozó lapokat (az INT 30h-n keresztül) is. Mivel a DS a 0-ra mutat, ezért az adatszegmens lévő számláló eléréséhez a 80000000h (2 GB) ofszetcímet kell használni.

A taszkhoz tartozó két verem mérete 4 KB lesz, és a 0. szintű verem a saját fizikai címén helyezkedik el (deszkriptor a GDT-ben), a 3. szintű verem pedig a 3 GB-on. Az utóbbi teszteléséhez néhány PUSH és POP utasítást célszerű végrehajtani, mivel az INT 30h csak a 0. szintű vermet használja.

task.inc

```
T1_Code Segment Para Public Use32
        Assume cs: T1_Code, ds:Nothing
T1_Code_Start Label Byte

L1:
    mov  edx,0518h              ; az XY koordináták (5,24)
    mov  al,2                   ; beállítása
    int  30h

    mov  edx,ds:[80000000h]     ; a számláló 2 Gb-nál
    dec  al
    int  30h                    ; kiírás

    inc  edx                    ; a számláló növelése
    mov  ds:[80000000h],edx

    pushad                      ; a 3. szintû verem használata
    popad

    Jump32_32 Gate_1,0          ; visszaugrás a hívó taszkra

    jmp  L1

T1_Code_Size Equ $-T1_Code_Start
T1_Code Ends

T1_Data Segment Para Public Use32
T1_Data_Start Label Byte

T1_Counter dd 0

T1_Data_Size Equ $-T1_Data_Start
T1_Data Ends

; A veremszegmensek
```

```
T1_Stack Segment Para Stack        ; 3. szintû vermem
T1_Stack_Size Equ 1000h
db T1_Stack_Size dup (?)
T1_Stack Ends

T1_Stack0 Segment Para Stack       ; 0. szintû verem
T1_Stack0_Size Equ 1000h
db T1_Stack0_Size dup (?)
T1_Stack0 Ends
```

Most tehát majdnem teljesen megszűnt a szegmensszintű védelem, a programhibák és egyebek kezelése a lapszintű védelmen keresztül történik. Ha például a T1 taszk tudni szeretné, hogy mi rejtőzik a 3333 MB-os memóriaterületen (D0500000h), akkor laphiba keletkezik (a CR2-be is ez a cím kerül).

Hogy megnézzük, valóban az 1 GB-on fut-e a T1 taszk, elhelyezhetünk egy INT 3 utasítást a kódszegmens legelejére, amely hatására kiíródnak a regiszterek. Ez alapján látható, hogy a CS-ben lévő szelektor 0007h (0. lokális deszkriptor, RPL = 3), az EIP-ben lévő érték pedig 40000001h, amely az 1 byte-os INT 3 utasítást követő ofszetcím.

3.3. Több taszk azonos címterületen

Kipróbálhatjuk a lapozást több taszkkal is, mégpedig úgy, hogy a taszkok címterülete azonos legyen. Most a szegmensdeszkriptorok báziscímét nem 0-ra állítjuk, hanem a szegmensek kezdetére. Ez azért hasznosabb, mivel így nem kell mindig hatalmas ofszetcímeket használni (bár néhány linker képes az ofszetcímek automatikus beállítására is).

A lokális deszkriptortáblázatok bevezetése óta minden taszk számára saját LDT-t alakítottunk ki. Erre most nincs igazán szükség, mivel a két taszk szegmensei ugyanazon a lineáris címeken lesznek, ezért a deszkriptoraik is megegyeznek. Egy közös LDT-ben (T12_LDT) helyezzük el tehát a kódszegmens (1 GB báziscím, 1 MB határ), az adatszegmens (2 GB báziscím, 1 MB határ), a veremszegmens (3 GB báziscím, 1 KB határ), és a T0-ra mutató taszkkapu deszkriptorait.

ldt.inc

```
PM_Data Segment

T12_LDT Label Qword
Desc Code_12,40000000h,-1,Code_Desc3+Seg_32,T12_LDT
Desc Data_12,80000000h,-1,Data_Desc3+Seg_32,T12_LDT
Desc Stck_12,0C0000000h,400h,Data_Desc3+Seg_32,T12_LDT
Desc Gate_12,Task_0,0,TaskGate_Desc3,T12_LDT
T12_LDT_Size Equ $-T12_LDT

PM_Data Ends
```

A határ 0fffffh-ra állításával az egyes szelektorokkal a szegmens tényleges méretén túli terület is megcímezhető, a lapozás viszont gondoskodik arról, hogy a hibás memóriaelérések kivételt okozzanak, mivel az elérhetetlen lapokban a P bit törölve lesz.

A GDT-ben helyezkedik el a közös LDT, a három taszk-állapotszegmens, és a 0. privilégiumszintű veremszegmensek deszkriptora.

gdt.inc

```
GDT_Seg Segment

Descriptor Task_0,0,TSS_Size,TSS_Desc
Descriptor Task_1,0,TSS_Size,TSS_Desc
Descriptor Task_2,0,TSS_Size,TSS_Desc

Descriptor LDT_12,0,T12_LDT_Size,LDT_Desc

Desc Stack0_1,0,T12_Stack0_Size,Data_Desc+Seg_32
Desc Stack0_2,0,T12_Stack0_Size,Data_Desc+Seg_32

GDT_Seg Ends
```

Bár a T1 és T2 taszkhoz tartozó lapok lineáris címe megegyezik, mégsem használhatjuk ugyanazokat a lapcímtárat és a laptáblázatokat a két taszk esetén, mivel az egyes lapok fizikailag más címen vannak (a bejegyzések így eltérnek). Esetleg a felügyelői lapok leképezését lehetne összevonni, mivel a két laptáblázat megegyezik (PageT1M és PageT2M). A T2 taszk szegmenseit is külön lapokra (T2_Copy) másoljuk majd át, hogy ne legyen átfedés a lapok és a szegmensek között, és hogy 4 KB-ra illeszkedjen minden lap.

page.inc

```
Page_Seg Segment Para Public Use32

; Táblázatok a T0 és a T1 taszkok számára

...

; Táblázatok a T2 taszk számára

PageD2              dd 1024 dup (?)     ; T2 lapcímtár

PageT2C             dd 1024 dup (?)     ; T2 laptáblázatai
PageT2D             dd 1024 dup (?)
PageT2S             dd 1024 dup (?)
PageT2M             dd 1024 dup (?)

T2_Copy             db 3*4096 dup (?)   ; T2 másolata

Page_Seg Ends
```

A védett módba kapcsolás után az EnablePages eljárással engedélyezzük a lapozást, amely a PG bit 1-re állítása előtt beállítja a 3 taszk laptáblázatait. Az InitPages_T2 eljárás a T2 taszk szegmenseit egyenként átmásolja a T2_Copy területre, majd beállítja a felhasználói és a felügyelői lapokat, majd a lapcímtárban elhelyezi a laptáblázatok címét.

```
InitPages_T2 Proc

   push ds es

   mov  ax,Flat32              ; ES = lineáris 0
   mov  es,ax
   mov  ds,ax

; A T2 taszk szegmenseinek átmásolása
; egy 4K-ra illeszkedõ területre

   lea  edi,T2_Copy            ; EDI = T2 másolat 0. lapja
   add  edi,ebx
   xor  esi,esi
   mov  si,T2_Code             ; ESI = T2 kódszegmensének
   shl  esi,4                  ; lineáris címe
   mov  ecx,T2_Code_Size ; a szegmens hossza
   rep  movsb                  ; átmásolás...

   lea  edi,T2_Copy+4096 ; EDI = T2 másolat 1. lapja
   add  edi,ebx
   xor  esi,esi
   mov  si,T2_Data             ; ESI = T2 adatszegmensének
```

```
    shl   esi,4                 ; lineáris címe
    mov   ecx,T2_Data_Size ; a szegmens hossza
    rep   movsb                 ; átmásolás...

    lea   edi,T2_Copy+8192 ; EDI = T2 másolat 2. lapja
    add   edi,ebx
    xor   esi,esi
    mov   si,T2_Stack           ; ESI = T2 veremszegmensének
    shl   esi,4                 ;  lineáris címe
    mov   ecx,T12_Stack_Size    ; a szegmens hossza
    rep   movsb                 ; átmásolás...

; A 3 szegmenshez tartozó laptáblázatok beállítása

    lea   edi,PageT2C           ; ES:EDI = a kódszegmens
                                ; táblázata
    add   edi,ebx               ; lineáris cím
    mov   eax,7                 ; jelenlévõ, írható
                                ; és felhasználói lap
    add   eax,offset T2_Copy
    add   eax,ebx               ; EAX = T2 kódszegmensének
                                ; lineáris címe
    mov   es:[edi],eax

    add   edi,4096              ; ES:EDI = az adatszegmens
                                ; táblázata
    add   eax,4096              ; a következõ lap ofszetcíme
    mov   es:[edi],eax

    add   edi,4096              ; ES:EDI = a veremszegmens
                                ; táblázata
    add   eax,4096              ; a következõ lap ofszetcíme
    mov   es:[edi],eax

; A 0-4 MB lineáris memória leképezése a fizikai
; memóriára, a felügyelõi szint számára

    lea   edi,PageT2M           ; ES:EDI = elsõ 4 MB táblázata
    add   edi,ebx               ; valódi 4K-s lapra igazítás
    mov   eax,3                 ; jelenlévõ, írható
                                ; és felügyelõi lapok.
    mov   ecx,1024              ; 1 táblázat (4M)
Init_T2:
    stosd
    add   eax,4096              ; a következõ lap ofszetcíme
    loop  Init_T2

; A Lapcímtár beállítása

    lea   edi,PageD2            ; A teljes lapcímtár
    add   edi,ebx
    mov   ecx,1024
```

```
    xor   eax,eax
    rep   stosd                   ; érvénytelenítése


    lea   edi,PageD2              ; ES:EDI = lapcímtár
    add   edi,ebx

    lea   eax,PageT2M             ; 0. bejegyzés
    add   eax,ebx
    add   eax,3
    mov   [edi],eax

    lea   eax,PageT2C             ; 256. bejegyzés
    add   eax,ebx
    add   eax,7
    mov   [edi+256*4],eax

    lea   eax,PageT2D             ; 512. bejegyzés
    add   eax,ebx
    add   eax,7
    mov   [edi+512*4],eax

    lea   eax,PageT2S             ; 768. bejegyzés
    add   eax,ebx
    add   eax,7
    mov   [edi+768*4],eax

    pop   es ds
    ret

InitPages_T2 Endp

PM_Code Ends
```

Az új taszk feladata hasonlít a T1 taszkéhoz: egy számláló értékét növeli, és ennek aktuális tartalmát az INT 30h megszakításon keresztül kiírja a képernyő egy adott pozíciójába.

task.inc

```
T2_Code Segment Para Public Use32
        Assume cs: T2_Code, ds: T2_Data
T2_Code_Start Label Byte

L2:
    mov   edx,0618h               ; az XY koordináták (6,24)
    mov   al,2                    ; 2. funkció
    int   30h                     ; megszakítás hívása

    mov   edx,T2_Counter; a számláló kiírása
    dec   al                      ; 1. funkció
```

```
    int  30h                    ; megszakítás hívása

    inc  T2_Counter             ; a számláló növelése

    Jump32_32 Gate_12,0; visszaugrás a hívó taszkra

    jmp  L2

T2_Code_Size Equ $-T2_Code_Start
T2_Code Ends


T2_Data Segment Para Public Use32
T2_Data_Start Label Byte

T2_Counter dd 0

T2_Data_Size Equ $-T2_Data_Start
T2_Data Ends

T12_Stack_Size  Equu 400h
T12_Stack0_Size Equ 400h

T2_Stack Segment Para Stack ; 2. taszk 3. szintû verme
db T12_Stack_Size dup (?)
T2_Stack Ends

T2_Stack0 Segment Para Stack; 2. taszk 0. szintû verme
db T12_Stack0_Size dup (?)
T2_Stack0 Ends
```

A két taszk TSS-eiben hasonló adatokat kell beállítani, ezért ezeket végezhetjük párhuzamosan is. Az EDX regiszter mutat az egyik TSS-re, az EBX regiszter pedig a másikra. Az FS és a GS szegmensregiszterekbe most 0-t állítunk be, és a GDT 0. deszkriptorával történő minden hivatkozás kivételt okoz. Tehát ha a taszkok használni szeretnék a GS és FS-t, akkor egy érvényes szelektort kell tölteniük ezekbe a regiszterekbe.

pm.inc

```
PM_Code Segment
       Assume cs: PM_Code, ds:PM_Data, es:PM_Data
PM_Main:

    call EnablePaging

    lea  esi,Ex3_Str
    call WriteStr
```

```
mov  dl,15
call SetColor

mov  ax,Task_0              ; az elsõ taszk kiválasztása
ltr  ax

lea  edx,TSS_0              ; DS:EDX =aktuális taszk TSS-e
mov  eax,CR3
mov  [edx].TSS_CR3,eax      ; a lapcímtár tárolása


lea  edx,TSS_1              ; DS:EDX = az 1. taszk TSS-e
lea  ebx,TSS_2              ; DS:EBX = a 2. taszk TSS-e

xor  eax,eax
mov  [edx].TSS_EFLAGS,eax; EFLAGS = 0
mov  [ebx].TSS_EFLAGS,eax
mov  [edx].TSS_EIP,eax              ; EIP = 0
mov  [ebx].TSS_EIP,eax

mov  [edx].TSS_FS,ax                ; FS, GS = 0
mov  [edx].TSS_GS,ax                ; (érvénytelenek)
mov  [ebx].TSS_FS,ax
mov  [ebx].TSS_GS,ax

mov  eax,Code_12+RPL3               ; a kódszegmensek
mov  [edx].TSS_CS,ax
mov  [ebx].TSS_CS,ax

add  eax,Data_12-Code_12            ; az adatszegmensek
mov  [edx].TSS_DS,ax
mov  [edx].TSS_ES,ax
mov  [ebx].TSS_DS,ax
mov  [ebx].TSS_ES,ax

add  eax,Stack_12-Data_12           ; a veremszegmensek
mov  [edx].TSS_SS,ax
mov  [ebx].TSS_SS,ax

mov  ax,Stack0_1                    ; a 0. szintû verem
mov  [edx].TSS_SS0,ax               ; szelektorai
mov  ax,Stack0_2
mov  [ebx].TSS_SS0,ax

mov  ax,T12_Stack_Size              ; veremmutatók
mov  [edx].TSS_ESP,ax
mov  [ebx].TSS_ESP,ax
mov  [edx].TSS_ESP0,ax
mov  [ebx].TSS_ESP0,ax

mov  ax,LDT_12                      ; az LDT deszkriptor
mov  [edx].TSS_LDTR,ax
```

```
    mov   [ebx].TSS_LDTR,ax

    mov   eax,CR3                          ; EAX = Page_Seg
    add   eax,offset PageD1                ; a T1 lapcímtárának
    mov   [edx].TSS_CR3,eax                ; tárolása
    add   eax,PageD2-PageD1                ; a T1 lapcímtárának
    mov   [ebx].TSS_CR3,eax                ; tárolása

    mov   ecx,10000h
PM_1:

    Jump32_32 Task_1,0                     ; ugrás az 1. taszkra
    Jump32_32 Task_2,0                     ; ugrás a 2. taszkra
    dec   ecx
    jnz   PM_1

    call DissablePaging

    ret


PM_Data  Segment

Ex3_Str  db '3. Példa: Két 3. szintû taszk azonos'
         db lineáris címen.',13,13
         db '1. Taszk számlálója:  [          ]',13
         db '2. Taszk számlálója:  [          ]',13,0

PM_Data Ends
```

Jelenleg a védelem elég hatásos, az alkalmazói szintű programok nem tehetnek kárt sem egymás, sem a felsőbb szintek memóriaterületén. Ha például a T1 taszk megpróbálja a T2 taszk adatszegmensét felülírni, akkor valójában a saját adatszegmensét teszi tönkre, mivel a két taszk szelektorainak számértéke megegyezik, a különbséget lényegében az eltérő lokális deszkriptortáblázatok jelentik.

Az sem jelentene igazán gondot, ha a két taszk deszkriptorait a GDT-ben helyezzük el. Ez esetben a T1 taszk elérheti a T2 adatszegmens szelektorát, és akár írhat is bele. Ekkor viszont újra a saját adatszegmensét teszi tönkre, mivel a két taszk adatai ugyanazon a lineáris címen helyezkednek el (2 GB).

3.4. Egy virtuális üzemmódú taszk elindítása

Ebben a példában egy virtuális módú taszkot indítunk el, mégpedig úgy, hogy a TSS-ben az EFLAGS regiszter VM bitjét 1-re állítjuk. Az IOPL értéke 3 lesz, így a virtuális módban érzékenynek számító utasítások végrehajtásakor nem keletkezik általános védelmi hiba. Ekkor a V86-os taszk valamivel gyorsabban fut (nincs szükség emulálásra), és a védett módú megszakításokat is meghívhatjuk közvetlenül virtuális módból. Az egyedüli hátrány, hogy az IF bit megváltoztatható, de mivel a megszakítások jelenleg úgysem keletkezhetnek (a PIC miatt), ezért ennek most nincs jelentősége.

A virtuális módú taszkból való kilépést a védett módú INT 31h-n keresztül valósítjuk meg. Ehhez mindössze az IDT 49. pozíciójában létre kell hozni egy 3. szintű megszakításkaput, amely egy olyan címre mutat, ahol visszakapcsolunk a felügyelői taszkra egy távoli ugrással.

idt.inc

```
IDT_Seg Segment

IDT Label QWord

; A kivételek kapui

    Int_Gate3 0,Code32,PM_Code:Exc_0
    ...

; A képernyõkezelés funkciók

    Int_Gate3 30h,Code32,PM_Code:Int_30

; Kilépés a virtuális módú taszkból

    Int_Gate3 31h,Code32,PM_Code:Int_31

IDT_Len Equ $-IDT

IDT_Seg Ends

PM_Code Segment

Int_31:

    Jump32_32 Task_0,0          ; visszaugrás a T0 taszkra

PM_Code Ends
```

A virtuális módú taszkkal kapcsolatban csak 2 globális deszkriptorra lesz szükség: a TSS deszkriptorra és a 0. szinthez tartozó veremszegmens deszkriptorára.

gdt.inc

```
Descriptor Task_0,0,TSS_Size,  TSS_Desc

Descriptor Task_1,0,TSS_1_Size,TSS_Desc
Descriptor Stack_V, 0,T1_Stack_Size,Data_Desc+Seg_32
```

A taszk regisztereiben valós módú értékeket helyezünk el: a CS:IP a kódszegmens (T1_Code) elejére mutat, az SS:SP a verem (T1_Stack) tetejére, a többi szegmensregiszter pedig az adatszegmens báziscímét tartalmazza. Beállítjuk az I/O engedélyező bittérkép címét is, ezáltal szükség esetén engedélyezhetjük, hogy a taszk bizonyos I/O portokhoz hozzáférjen. Az új taszkot tehát egy távoli ugrással indítjuk el, és az INT 31h megszakítással térünk vissza.

pm.inc

```
PM_Code Segment
       Assume cs: PM_Code, ds:PM_Data, es:PM_Data
PM_Main:

    lea  esi,Ex4_Str
    call WriteStr

    mov  ax,Task_0              ; az elsõ taszk kiválasztása
    ltr  ax

    lea  edx,TSS_1

    mov  [edx].TSS_CS,T1_Code; CS:IP=szegmens:ofszet
    mov  [edx].TSS_EIP,0

    mov  ax,T1_Data             ; 16 bites báziscím
    mov  [edx].TSS_DS,ax        ; a szegmensregiszterekbe
    mov  [edx].TSS_ES,ax
    mov  [edx].TSS_FS,ax
    mov  [edx].TSS_GS,ax

    mov  [edx].TSS_SS,T1_Stack          ; valós módú verem
    mov  [edx].TSS_ESP,T1_Stack_Size
    mov  [edx].TSS_SS0,Stack_V          ; védett módú verem
    mov  [edx].TSS_ESP0,T1_Stack0_Size
```

```
    mov   [edx].TSS_EFLAGS,23002h  ; VM=1, IOPL=3

    mov   [edx].TSS_IOBASE,IOBASE_1; I/O térkép címe

    Jump32_32 Task_1,0             ; ugrás a V86-os taszkra

    ret


PM_Code Ends


PM_Data Segment

Ex4_Str db '4. Példa: Virtuális üzemmódú taszk '
        db 'elindítása.',13,0

PM_Data Ends
```

A virtuális mód működését kipróbálhatjuk, ha az ES regiszterbe a képernyő valós módú báziscímét töltjük be, és így írunk a képernyőre. Hogy megnézzük, tényleg a védett módú megszakítás-kezelő hívódik meg az IOPL = 3 esetén, a képernyőre kiírunk egy hexadecimális számot is az INT 30h funkciói segítésével.

task.inc

```
T1_Code Segment Para Public Use16
        Assume cs: T1_Code, ds: T1_Data

; Képernyõre írás közvetlenül

    mov   ax,0b800h
    mov   es,ax
    mov   di,5*160              ; ES:DI = a képernyõ 5. sora

    lea   si,T1_Str             ; a szöveg kiírása
    mov   ah,7                  ; szürke színnel
L1:
    lodsb
    or    al,al                 ; ha 0, a szöveg vége
    je    L2
    stosw
    jmp   L1

; Képernyõre írás az INT 30h védett módú
; megszakításon keresztül

L2:
    mov   al,2
```

```
    mov  dx,600h                ; 6. sor eleje
    int  30h                    ; megszakítás hívása

    dec  al
    mov  dx,1234h
    int  30h                    ; DX kiírása

    pushf                       ; IOPL=3, most mindent szabad
    cli
    sti
    popf

V86_Exit:
    int 31h                     ; kilépés a V86-os taszkból


T1_Code Ends

T1_Data Segment Para Public Use16

T1_Str  db 'Képernyõre írás V86 módban.',0

T1_Data Ends


; A veremszegmensek

T1_Stack Segment Para Stack           ; 3. szintû verem
T1_Stack_Size Equ 400h
db T1_Stack_Size dup (?)
T1_Stack Ends

T1_Stack0 Segment Para Stack          ; 0. szintû verem
T1_Stack0_Size Equ 400h
db T1_Stack0_Size dup (?)
T1_Stack0 Ends
```

A fentiek tehát az IOPL = 3 esetre vonatkoznak, vagyis ekkor a virtuális módú taszk akadály nélkül végrehajthatja az érzékeny utasításokat. A legtöbb esetben ez nem biztonságos, ekkor a taszkok IOPL értékét 3-nál kisebbre kell állítani, és az egyes utasításokat emulálni kell.

3.5. Valós módban 4 GB memória elérése

Az összes eddigi példában, mielőtt visszakapcsoltunk volna valós módba, minden szegmensregiszterbe 64 KB méretű szegmensek szelektorát töltöttuk be. Most azt nézzük meg, hogy mi történik, ha úgy kapcsolunk vissza védett módból, hogy ezt nem végezzük el.

A szegmensregiszterek nem csak a szelektor részből állnak, hanem tartalmaznak egy láthatatlan részt is, ahol az adott szegmens báziscíme, mérete és elérési joga található. Valós módban egy szegmensregiszter feltöltésekor csak a báziscím változik meg, a szegmens mérete például változatlan marad. Ez a tulajdonság felhasználható arra, hogy megszabaduljunk a valós mód 64 KB-os korlátjától, és egy módosított szegmensregiszterrel akár 4 GB-ot is megcímezhessünk

Ennek az érdekes módnak már sokféle nevet adtak (Real Flat Mode, Unreal Mode, Real Big Mode, Voodoo Mode, stb.), de mindegyik elnevezés ugyanarra utal: a teljes 4 GB-os memóriaterület közvetlen elérése valós módban.

Ez az állapot tehát úgy valósítható meg, ha átkapcsolunk védett módba, és a szegmensregiszterekbe 4 GB méretű szegmensek szelektorát töltjük be, majd visszakapcsolunk valós módba anélkül, hogy megváltoztatnánk a szegmensregiszter határát. Ekkor a láthatatlan rész továbbra is a 4 GB-os határt tartalmazza, és ettől kezdve 32 bites ofszetcímekkel elérhető a teljes memória. Ekkor nem fog 13. kivételt okozni valós módban, ha 64 KB-nál nagyobb ofszetcímeket használunk.

A GDT-ben csak a 4 GB-os adatszegmens deszkriptorát kell definiálni (és a nulla deszkriptort), és a GDTR-t

gdt.inc

```
RealSeg Segment

GDT Label Qword

Descriptor NullSel,0,0,0     ; Nulla Deszkriptor
Descriptor Real4G,0,0ffffh,0c092h; 32 bites 4G méretû
                             ; adatszegmens

GDT_Len     Equ $-GDT        ; a GDT hossza

GDT_Value   Label Pword      ; LGDT utasítás paramétere
GDT_Limit   dw GDT_Len       ; a GDT hossza
GDT_Base    dd offset GDT    ; a GDT címe


;
; InitGDT: A GDTR inicializálása
```

```
InitGDT Proc

    mov  eax,cs                 ; eax = RealSeg
    shl  eax,4                  ; eax = báziscím
    add  GDT_Base,eax           ; a GDT_Base fizikai címmé
                                ; alakítása

    lgdt GDT_Value              ; a GDTR beállítása

    ret

InitGDT Endp


RealSeg Ends
```

Most csak az ES méretét változtatjuk meg a védett módba kapcsolás után a Real4G szelektorral, természetesen a többi szegmensregisztert is módosíthatjuk hasonló módon. A teszteléshez a képernyő felső sorát teleírjuk 'A' betükkel, mégpedig a 0b8000h címen, és nem a 0b800h-n.

example5.asm

```
.386p

RealSeg Segment Para Public Use16
       Assume  cs:RealSeg, ds:RealSeg

Real_Start:
    cli                         ; megszakítások letiltása

    push cs                     ; ds = RealSeg
    pop  ds

    call Check386               ; 386-os ellenõrzése
    call CheckV86               ; V86 mód ellenõrzése

    mov  ax,3                   ; képernyõtörlés
    int  10h

    mov  ah,9                   ; szöveg kiírása
    lea  dx,Ex5_Str
    int  21h

    call InitGDT                ; GDT beállítása

    mov  eax,CR0                ; átkapcsolás védett módba
    or   al,1
    mov  CR0,eax                ; PE bit=1
    jmp  $+2
```

```
    mov  ax,Real4G              ; 0 bázisú, 4Gb méretû
    mov  es,ax                  ; szelektor az ES-be

    mov  eax,CR0                ; visszakapcsolás valós módba
    and  al,not 1
    mov  CR0,eax                ; PE bit=0
    jmp  $+2


    mov  eax,'AAAA'             ; 2 színes A betû kiírása
    mov  ecx,40                 ; egy teljes sor
    mov  edi,0b8000h            ; az ofszet > 64 KB !
L1:
    mov  es:[edi],eax           ; 32 bites ofszet használata
    add  edi,4                  ; következö 2 karakter
    loop L1

    mov  ah,4ch                 ; kilépés
    int  21h

Exit:
    mov  ah,9
    int  21h                    ; string kiírása
    mov  ah,4ch
    int  21h                    ; kilépés


Ex5_Str   db 10,13,10,13
          db '5. példa: 4 GB memória elérése valós '
          db  módban.',10,13,36

RealSeg Ends

StackSeg Segment Stack Use16          ; 1K valós módú verem
db  400h dup (?)
StackSeg Ends

End Real_Start
```

A következő fejezetben az XMS-ről lesz szó, amely többek között a kiterjesztett memória lefoglalásában segít. A lefoglalt memóriát használhatjuk védett módban, vagy akár a most tárgyalt technikával valós módban is.

III. RÉSZ

1. Extended Memory Specification (XMS)

1.1. Az XMS driver ellenőrzése

Ahhoz, hogy megállapítsuk, az XMS driver elérhető-e, az INT 2Fh megszakítás AX = 4300h funkcióját kell meghívni. Ha az AL regiszter értéke visszatéréskor 80h, akkor az XMS driver installálva van.

```
mov   ax,4300h              ; az XMS installálva van?
int   2fh
cmp   al,80h
jne   XMS_Error             ; nincs.
```

Az XMS driver felismerhető az XMSXXXX0 eszközmeghajtó jelenlétének vizsgálatával is, bár a fenti módszer sokkal egyszerűbb.

1.2. Az XMS funkciók

Az XMS driver funkcióit egy FAR CALL utasítással lehet meghívni. Az INT 2Fh, AX = 4310h megszakítás segítségével ezt a távoli címet megkaphatjuk az ES:BX regiszterekben.

Az XMS driver címének lekérdezése		
Be	AX	4310h
Ki	ES:BX	az XMS driver címe (szegmens:ofszet)

A driver meghívásakor az AH tartalmazza a funkció sorszámát, visszatéréskor pedig az AX regiszter jelzi, hogy a funkció sikerült-e (AX = 1) vagy nem (AX = 0). Bármilyen hiba esetén a hibakód a BL regiszterben található (a lehetséges hibakódok egy későbbi fejezetben szerepelnek). Nem

igazán célszerű az AX regiszterben tárolni a funkció eredményét (sikerült vagy nem sikerült), hiszen ez akár egyetlen biten is elférne (pl. a BL legfelső bitjén).

Az XMS funkciók (ld. 1.1. ábra) lényegében az alsó 640 KB feletti memória kezelésével foglalkoznak. A következők tartoznak ide:

- a 640 KB és az 1 MB közötti terület – Upper Memory Block (UMB)
- az 1MB feletti 64 KB méretű terület . High Memory Area (HMA)
- az 1088K feletti kiterjesztett memória – Extended Memory (EMB)
- Az A20 kezelése – az 1 MB feletti terület engedélyezése és letiltása

Sorszám	Funkció megnevezése
0	Verziószám lekérdezése
1	HMA lefoglalása
2	HMA felszabadítása
3	A20 globális engedélyezése
4	A20 globális letiltása
5	A20 lokális engedélyezése
6	A20 lokális letiltása
7	A20 lekérdezése
8	Szabad EMB lekérdezése
9	EMB lefoglalása
A	EMB felszabadítása
B	EMB mozgatása
C	EMB lezárása
D	EMB megnyitása
E	EMB azonosító információjának lekérdezése
F	EMB átméretezése
10	UMB lefoglalása
11	UMB felszabadítása
12	UMB átméretezése
88	Szabad EMB lekérdezése
89	Kiterjesztett memória foglalása
8E	Kiterjesztett memória azonosító lekérdezése
8F	Kiterjesztett memória átméretezése

1.1. ábra Az XMS driver funkciói

1.3. Az XMS driver információ lekérdezése

A 0. funkcióval megtudhatjuk az XMS driver verziószámát, és hogy az 1 MB feletti 64 KB méretű terület használható-e.

Verziószám lekérdezése		
Be	AH	0
	AX	XMS verziószám (BCD)
Ki	BX	Revíziószám
	DX	Ha a HMA létezik, akkor 1 Ha nem létezik, akkor 0

1.4. A HMA-val kapcsolatos funkciók

A HMA-nak a védett módú programok számára kevés jelentősége van, mivel túl kicsi a mérete. Valós módban is többnyire csak a TSR-ek használják, mivel ezen az 1 MB feletti területen is futtatható program valós módban (ha az A20 engedélyezve van).

HMA lefoglalása		
	AH	1
Be	DX	Szükséges byte-ok száma, ha a hívó egy TSR vagy egy eszközmeghajtó, és 0FFFFh alkalmazói program esetén
Ki	AX	1, ha sikerült, 0, ha nem

HMA felszabadítása		
Be	AH	2
Ki	AX	1, ha sikerült, 0, ha nem

1.5. Az A20 funkciók

Az A20 címvonal engedélyezése a feltétele annak, hogy használhassuk az 1 MB feletti memóriát. Az engedélyezés és a letiltás történhet globálisan és lokálisan is, a globális engedélyezés általában a HMA-t is használó prog-

ramoknál szükséges, a kiterjesztett memória használatához elég a lokális engedélyezés.

A20 globális engedélyezése		
Be	AH	3
Ki	AX	1, ha sikerült, 0, ha nem

A20 globális letiltása		
Be	AH	4
Ki	AX	1, ha sikerült, 0, ha nem

A20 lokális engedélyezése		
Be	AH	5
Ki	AX	1, ha sikerült, 0, ha nem

A20 lokális letiltása		
Be	AH	6
Ki	AX	1, ha sikerült, 0, ha nem

A20 állapotának lekérdezése		
Be	AH	7
Ki	AX	1, ha engedélyezve van, 0, ha nem

1.6. Kiterjesztett memória kezelése

Az XMS legfontosabb funkciói a kiterjesztett memóriával kapcsolatosak. Megtalálhatóak köztük az általános funkciók: a foglalás, a felszabadítás, és az átméretezés. A 8. funkcióval a legnagyobb egybefüggő terület, illetve a teljes szabad kiterjesztett memória mérete kérdezhető le. A 9. funkcióval egy EMB (Extended Memory Block) foglalható le, és ha ez sikerült, akkor egy azonosítót (handle) kapunk vissza, amely segítségével a blokkon egyéb műveleteket végezhetünk (felszabadítás, információ lekérdezése, átméretezés).

Szabad kiterjesztett memória méretének lekérdezése		
Be	AH	8
Ki	AX	a legnagyobb EMB mérete KB-ban
	DX	teljes szabad kiterjesztett memória mérete KB-ban

EMB lefoglalása		
Be	AH	9
	DX	szükséges memória mérete KB-ban
Ki	AX	1, ha sikerült, 0, ha nem
	DX	azonosító

EMB felszabadítása		
Be	AH	Ah
	DX	azonosító
Ki	AX	1, ha sikerült, 0, ha nem

EMB azonosító lekérdezése		
Be	AH	Eh
	DX	azonosító
	AX	1, ha sikerült, 0, ha nem
Ki	DX	a blokk mérete kbyte-okban
	BH	a blokk lezárásainak száma
	BL	a szabad azonosítók száma

EMB átméretezése		
	AH	Fh
Be	DX	a blokk azonosítója
	BX	a blokk új mérete kbyte-okban
Ki	AX	1, ha sikerült, 0, ha nem

Lehetőség van az adatok másolására is, a kiterjesztett memória és az alsóbb területek között, valamint két lefoglalt kiterjesztett memóriablokk között. A funkciónak egy 16 byte-os struktúra címét kell megadni, ahol szerepel a másolandó terület mérete byte-okban, és hogy honnan hova akarunk másolni. A címek 48 bites mutatók, EMB esetén a 16 bites azonosítóból és a 32 bites ofszetcímből áll, az 1 MB alatti terület esetén az azonosítót 0-ra kell állítani, és a valós módú szegmens:ofszetcímet kell megadni a 32 bites címen.

EMB mozgatása		
Be	AH	Bh
	DS:SI	mozgatáshoz szükséges adatok címe
Ki	AX	1, ha sikerült, 0, ha nem

A mozgatáshoz szükséges adatok	
Az átmásolandó byte-ok száma	32 bit
A forrásblokk azonosítója, ahonnan másolunk	16 bit
A forrásblokk címe	32 bit
A célblokk azonosítója, ahová másolunk	16 bit
A célblokk címe	32 bit

Elméletileg az XMS driver az egyes blokkokat szabadon áthelyezheti a memóriában (ha sok kis lyuk keletkezett), hiszen a lefoglaláskor csak egy azonosítót kapunk vissza. Ha ezt meg akarjuk tiltani, akkor rögzíthetjük (lock) a blokkot a 12. funkcióval. Ennél viszont sokkal nagyobb jelentősége a funkciónak, hogy így megkapjuk a lefoglalt blokk pontos helyét is a memóriában. Ezután átkapcsolhatunk védett módba, és a 32 bites lineáris cím segítségével használhatjuk az 1 MB feletti területet. Valós módba visszakapcsolva pedig a 13. funkció (unlock) meghívása után felszabadíthatjuk a blokkot. Ez sokkal szebb megoldás, mintha védett módban egyszerűen csak felülírjuk a kiterjesztett memóriát.

EMB lezárása		
Be	AH	Ch
	DX	a lezárandó blokk azonosítója
Ki	AX	1, ha sikerült, 0, ha nem
	DX:BX	a blokk 32 bites fizikai címe

EMB megnyitása		
Be	AH	Dh
	DX	a megnyitandó blokk azonosítója
Ki	AX	1, ha sikerült, 0, ha nem

1.7. Az UMB-vel kapcsolatos funkciók

A felső memóriaterülettel is elvégezhetőek a szokásos műveletek: a memóriafoglalás, a -felszabadítás, és -átméretezés. Egy UMB (Upper Memory Block) lefoglalásakor egy 16 bites szegmenset kapunk, amely ugyancsak felhasználható védett módban (16-al való szorzással lineáris címet kaphatunk).

UMB lefoglalása		
Be	AH	10h
	DX	a szükséges memória mérete paragrafusokban
	AX	1, ha sikerült, 0, ha nem
Ki	BX	a lefoglalt blokk szegmenscíme
	DX	a blokk mérete, ha a funkció sikerült legnagyobb szabad UMB mérete, ha nem sikerült

UMB felszabadítása		
Be	AH	11h
	DX	a blokk szegmenscíme
Ki	AX	1, ha sikerült, 0, ha nem

UMB átméretezése		
	AH	12h
Be	BX	az új méret paragrafusokban
	DX	a blokk szegmenscíme
Ki	AX	1, ha sikerült, 0, ha nem

1.8. 32 bites kiterjesztett memória funkciók

Az XMS 3.0 verziónál megjelent néhány új funkció is, amelyek már 32 bites paramétereket is támogatnak. Ezek bevezetésére azért volt szükség, mivel a 0–12h közötti funkciók csak 16 bites paramétereket használtak (tekintettel a 286-os processzorra), és így például a 9. funkcióval csak 64 MB-nál kisebb területeket lehetett lefoglalni (65535 * 1 KB). Hatalmas memóriablokkok lefoglalásakor tehát ezeket a funkciókat célszerű használni.

Szabad kiterjesztett memória méretének lekérdezése		
Be	AH	88h
	EAX	a legnagyobb EMB mérete KB-ban
Ki	EDX	teljes szabad kiterjesztett memória mérete KB-ban
	ECX	a fizikai memória teteje, az utolsó byte lineáris címe

Kiterjesztett memória foglalása		
Be	AH	89h
	EDX	szükséges memória mérete KB-ban
Ki	AX	1, ha sikerült, 0, ha nem
	DX	azonosító

Kiterjesztett memória azonosító lekérdezése		
Be	AH	8Eh
	DX	azonosító
	AX	1, ha sikerült, 0, ha nem
Ki	EDX	a blokk mérete kbyte-okban
	BH	a blokk lezárásainak száma
	CX	a szabad azonosítók száma

Kiterjesztett memória átméretezése		
	AH	8Fh
Be	DX	a blokk azonosítója
	EBX	a blokk új mérete kbyte-okban
Ki	AX	1, ha sikerült, 0, ha nem

1.9. Az XMS hibakódok

Ha az XMS funkciók hívása valamilyen hiba miatt nem sikerül (vagyis visszatéréskor az AX = 0), akkor a BL regiszter tartalmazza a hibakódot, amelynek a legfelső bitje mindig 1.

Hibakód	Megnevezés
80	érvénytelen funkció
81	VDISK eszköz jelen van
82	A20 hiba
8E	általános driver hiba
8F	helyrehozhatatlan hiba
90	HMA nincs jelen
91	HMA már használatban van
92	a kért HMA memória mérete kevés (1. funkció esetén)
94	A20 továbbra is engedélyezve van (4. és 6. funkciók)
A0	nincs több szabad kiterjesztett memória
A1	nincs több szabad azonosító
A2	érvénytelen azonosító
A3–A9	érvénytelen paraméter a mozgatáshoz (11. funkció)
AA	terület nincs lezárva
AB	terület le van zárva
AC	túl sok lezárás
AD	nem sikerült egy területet lezárni
B0	kis UMB terület
B1	nincs szabad UMB terület
B2	érvénytelen UMB szegmens

1.2. ábra Az XMS driver hibakódjai

1.10. Példa a kiterjesztett memória használatára védett módban

Az 1 MB feletti területet használó DOS extenderek, vagy egyéb védett módú programok esetén nem ajánlatos önkényesen felülírni a memóriát, mivel előfordulhat, hogy már más program is (pl. disk cache) használja ugyanazt a területet. Az XMS funkciók használatával ez a probléma elkerülhető.

Bár az XMS-t elsősorban a valós módú alkalmazások számára fejlesztették ki, a védett módú programok számára is megkönnyítik a memóriakezelést az XMS funkciók. Egyrészt az XMS driveren keresztül engedélyezhető az A20, így a kiterjesztett memória is elérhetővé válik, anélkül, hogy minden egyes programban közvetlenül a billentyűzetvezérlőn keresztül kel-

lene engedélyezni. Másrészt a memóriafoglalások során megkaphatjuk a lefoglalt memória lineáris címét, amelyet védett módban minden korlátozás nélkül használhatunk, vagyis egyetlen szegmensregiszterrel megcímezhetővé válik a több MB-os terület is. Védett módban a kiterjesztett memória már nem csak adatok tárolására használható, mint valós módban, hanem programok futtatására is alkalmas.

Ebben a példában lefoglaljuk a teljes szabad kiterjesztett memóriát, átkapcsolunk védett módba, majd ezt az egész memóriablokkot felülírjuk. A védett módba kapcsolás előtt ellenőrizni kell a processzort (legalább 386-os, és valós módban legyen), majd inicializálni kell a GDT-t és a GDTR-t.

A kiterjesztett memória foglalása előtt ellenőrizni kell, hogy elérhető-e az XMS driver. Ha nem, akkor egy hibaüzenettel kilépünk. Ha az XMS driver jelen van, akkor a driver címének lekérdezése után már hívhatjuk az egyes funkciókat. Először engedélyezzük az A20-at, majd lekérdezzük a szabad memóriát, és lefoglaljuk az egészet. Ekkor tehát csak egy azonosítót kapunk, a lineáris cím lekérdezéséhez a 12. funkciót is meg kell hívni. Ha ez sikerült, akkor már használhatjuk is a memóriaterületet.

A védett módú program befejezésével a valós módba kapcsolás után fel kell szabadítani a blokkot az elmentett azonosító alapján.

A következő két eljárás ezeket a feladatokat végzi el. A védett módba kapcsolás előtt az InitXMS-t kell meghívni, amely visszaadja a blokk lineáris címét (EDI) és méretét (ECX), a visszakapcsolás után pedig a FreeXMS-t kell meghívni, amely felszabadítja a memóriát. Ha hiba történik, akkor egy hibaüzenettel kilépünk.

```
RealSeg Segment

InitXMS Proc

    mov   ax,4300h              ; XMS jelenlétének vizsgálata
    int   2fh

    lea   dx,Str_NoXMS
    cmp   al,80h
    jne   XMS_Error             ; hiba, ha nincs XMS

    push es
    mov   ax,4310h              ; driver címének lekérdezése
```

```
    int   2fh
    mov   XMS_Address,bx        ; ofszet cím tárolása
    mov   XMS_Address+2,es      ; szegmens cím tárolása
    pop   es

    mov   ah,5                  ; A20 engedélyezése
    call  XMS_Driver

    lea   dx,Str_A20
    cmp   ax,0
    je    XMS_Error             ; hibaüzenet, ha nem sikerült

    mov   ah,8                  ; szabad ext. mem. lekérdezése
    call  XMS_Driver

    mov   XMS_Size,ax           ; a méret tárolása

    mov   dx,ax
    mov   ah,9                  ; az egész szabad terület
    call  XMS_Driver            ; lefoglalása

    cmp   ax,1                  ; sikerült?
    je    InitXMS_1

    lea   dx,Str_Alloc          ; nem sikerült a foglalás
    jmp   XMS_Error

InitXMS_1:
    mov   XMS_Handle,dx         ; a handle tárolása

    mov   ah,0Ch
    call  XMS_Driver            ; lineáris cím lekérdezése

    cmp   ax,1                  ; sikerült?
    je    InitXMS_2

    lea   dx,Str_Lock           ; nem sikerült.
    jmp   XMS_Error

InitXMS_2:

    mov   XMS_Linear,bx         ; a lineáris cím tárolása
    mov   XMS_Linear+2,dx

    mov   di,dx
    shl   edi,16
    mov   di,bx                 ; EDI = a lineáris cím

    xor   ecx,ecx
    mov   cx,XMS_Size           ; ECX = a blokk mérete
    shl   ecx,10                ; byte-okban
```

```
    ret

InitXMS Endp

FreeXMS Proc

    mov  ah,0Dh
    mov  dx,XMS_Handle
    call XMS_Driver

    mov  ah,0Ah                ; A blokk felszabadítása
    call XMS_Driver

    ret

FreeXMS Endp


XMS_Error Proc

    mov  ah,9                  ; hibaüzenet kiírása
    int  21h
    mov  ah,4ch                ; kilépés
    int  21h

XMS_Error Endp

XMS_Driver     Label Dword
XMS_Address    dw 0,0          ; Az XMS driver címe
XMS_Size       dw 0            ; A blokk mérete (Kbyte)
XMS_Handle     dw 0            ; A blokk azonosítója
XMS_Linear     dw 0,0          ; A blokk lineáris címe

; Hibaüzenetek

Str_NoXMS   db 'Az XMS driver nem található.',10,13,36
Str_A20     db 'Az A20 engedélyezése nem sikerült.'
            db 10,13,36
Str_Alloc   db 'A memóriafoglalás nem sikerült.'
            db 10,13,36
Str_Lock    db 'A lineáris cím lekérdezése nem'
            db sikerült',10,13,36

Ends
```

A lefoglalt memóriablokk elérhető egy 0 bázisú, 4 GB méretű szegmenssel is, vagy egy külön deszkriptort is beállíthatunk a blokk báziscímének és méretének megfelelően (ez utóbbi biztonságosabb). Az alábbi néhány utasítással például felülírhatjuk az egész kiterjesztett memóriablokkot.

```
; EDI = XMS blokk lineáris címe, ECX = a blokk mérete

    mov   ax,Flat32              ; ES:EDI = blokk
    mov   es,ax

    shr   ecx,2                  ; ECX = méret duplaszavakban
    mov   eax,'!SMX'
    rep   stosd                  ; a blokk feltöltése 'XMS!'-el
```

A kiterjesztett memóriába másolhatunk akár egy (vagy több) 32 bites programot is, és onnan futtathatjuk. A lapozás engedélyezésével és virtuális memória kialakításával pedig a lefoglalt memória mérete meg is többszörözhető.

A következő rész a DPMI-ről szól, amely már nem csak a memóriakezelésre alkalmas, mint az XMS, hanem hasznos védett módú funkciókat tartalmaz, mint például a deszkriptorok, és a megszakítások kezelése.

2. DOS Protected Mode Interface (DPMI)

A DPMI a legtöbb operációs rendszernél (Windows, OS/2, Linux, stb.) elérhető, és sok DOS extender is ehhez igazítja az egyes funkcióit. A DPMI tehát a lehetővé teszi, hogy a DOS alatt futó programok is hozzáférjenek a védett módú szolgáltatásokhoz, deszkriptorokhoz, kiterjesztett memóriához, megszakításokhoz, továbbá az üzemmódok közötti kommunikációt is megvalósítja.

A DPMI szolgáltatásokat védett módban, az INT 31h megszakításon keresztül lehet igénybe venni. Az egyes programok kezdetben valós vagy virtuális módban futnak, és az alkalmazásoknak kell a DPMI kiszolgáló hívásával védett módba kapcsolni.

A legtöbb C, Pascal, és egyéb fordító is alkalmas DPMI funkciókat használó programok készítésére. Erre szolgál például a Borland termékek esetében a DPMI szerver és RunTime Manager, amelynek 16 bites (DPMI16BI.OVL, RTM.EXE) és a 32 bites (DPMI32VM.OVL, 32RTM.EXE) változata is van.

2.1. Átkapcsolás védett módba a DPMI segítségével

A DPMI kiszolgáló ellenőrzéséhez és a védett módba kapcsolás távoli címének lekérdezéséhez az INT 2Fh valós módú megszakítás AX = 1687h funkcióját kell meghívni. A visszaadott regiszterek értéke a következő:

Védett módba kapcsolás címének lekérdezése	
AX	Ha 0, akkor a funkció sikerült Ha nem 0, akkor nincs DPMI kiszolgáló
BX	Ha a 0. bit = 0, a 32 bites programok nincsenek támogatva Ha a 0. bit = 1, a 32 bites programok támogatva vannak
CL	a processzor típusa (3 = 386, 4 = 486, stb.)

DX	a DPMI verziószám (DH,DL)
SI	szükséges memória mérete a DPMI számára (paragrafusban)
ES:DI	a védett módba lépés címe (szegmens:ofszet)

A védett módba kapcsoláshoz tehát meg kell hívni (FAR CALL) az ES:DI-ben lévő címet, az AX 0. bitjével jelezve a program típusát. (0 = 16 bites, 1 = 32 bites program). Vannak olyan DPMI-k, amelyek mind a két típusú programot kezelni tudják, és vannak olyanok is, amelyek csak a 32, vagy csak a 16 bites programok futtatását engedélyezik.

Ha az SI nem 0, akkor előtte memóriát is kell foglalni a kiszolgáló számára, például a DOS INT 21h, 48h funkciójával, és a lefoglalt memória címét az ES-be kell rakni.

Az eljárás hívása után a Carry 0, ha sikerült átkapcsolni védett módba, és 1, ha nem sikerült (ekkor továbbra is valós módban vagyunk).

A védett módba kapcsolás után az általános regiszterek változatlanok, és a szegmensregiszterek pedig az alábbi értéket tartalmazzák:

CS	16 bites kódszegmens szelektor, 64 KB határ
DS	adatszegmens szelektor, 64 KB határ
SS	veremszegmens szelektor, 64 KB határ
ES	a PSP szelektora, 256 byte határ
GS	0
FS	0

A deszkriptorok báziscíme megegyezik a korábbi valós módú báziscímmekkel. A CS-ben lévő szelektor akkor is 16 bites lesz, ha az AX az átkapcsoláskor 32 bites programot jelez. Ez esetben le kell foglalni egy új deszkriptort, beállítani a báziscímet, a határt, és a kódszegmens típusát, majd átadni a vezérlést az új szelektorra.

A védett módú programból a DOS-hoz hasonlóan az INT 21h, 4Ch funkcióval lehet kilépni. Ezt közvetlenül védett módban meg lehet hívni, nem kell valós módba visszakapcsolni.

A következő példán látható, hogy mennyire könnyen lehet a DPMI segítségével átkapcsolni védett módba, és vissza. Az egyszerűség kedvéért a DPMI cím hívásakor nem egy dinamikusan lefoglalt memóriaterületet adunk át az ES-ben, hanem egy 64 KB méretű külön szegmenset. (Ez egyébként sem foglalja a helyet az EXE file-ban.)

```
.386
.model small
.stack

DPMI    Segment Para Public Use16 'CODE'
   Assume cs:DPMI, ds:DPMI

SwitchPtr       dd 0        ; a DPMI módváltás távoli címe
Str_DPMI        db 'DPMI nem található.',10,13,'$'
Str_Switch      db 'Nem sikerült átkapcsolni védett '
                db 'módba.',10,13,'$'

Start:
   push cs
   pop  ds

   mov  ax,3                ; képernyõtörlés, stb.
   int  10h

   mov  ax,1687h            ; a DPMI host címének
   int  2fh                 ; lekérdezése
   or   ax,ax
   jnz  NoDPMI              ; nincs DPMI

   mov  word ptr SwitchPtr,di        ; ofszet cím
   mov  word ptr SwitchPtr+2,es      ; szegmens cím

   mov  ax,DPMI2            ; adatszegmens a host számára
   mov  es,ax
   xor  ax,ax               ; 0. bit: 16 bites program
   call SwitchPtr           ; átkapcsolás védett módba
   jc   NoSwitch            ; nem sikerült...

ProtMode:
   nop                      ; most védett módban vagyunk

Exit:
   mov  ax,4c00h            ; a szokásos kilépés
   int  21h

NoDPMI:                     ; hibaüzenetek
   lea  dx,Str_DPMI
   jmp  Error
NoSwitch:
```

```
    lea   dx,Str_Switch
Error:
    mov   ah,9
    int   21h
    jmp   Exit


DPMI      Ends

DPMI2 Segment                   ; privát adatszegmens
      db 0ffffh dup (?)         ; a DPMI host számára
DPMI2 Ends

End Start
```

2.2. A DPMI funkciók hívása

A védett módba kapcsolás után már használhatóak a DPMI szolgáltatások is, amelyek a deszkriptorokkal, a memóriakezeléssel, és a lapozással, stb. kapcsolatosak.

A DPMI funkciókat az INT 31h megszakításon keresztül lehet elérni. Ezek a funkciók a carry flagban jelzik az eredményt. Ha 0, akkor sikerült, ha 1, akkor pedig valamilyen hiba történt, és az AX tartalmazza a hibakódot (ld. a DPMI hibakódok című fejezetet). Mivel ez az összes funkcióra igaz, ezért ez nem szerepel külön minden egyes funkció ismertetésénél.

Néhány esetben zavaró, hogy a 32 bites paramétereket két 16 bites regiszterben kell átadni, illetve úgy kapjuk vissza, még akkor is, ha a program 32 bites. Például a memóriafoglalásnál a 32 bites azonosítót (handle) az SI:DI regiszterekben kapjuk meg, pedig a 16 bites (vagy akár a virtuális üzemmódban futó) programok is akadály nélkül használhatják a 32 bites regisztereket is.

2.3. Lokális deszkriptorok kezelése

A DPMI kiszolgálók alatt az egyes programok deszkriptorai külön lokális deszkriptortáblázatokban vannak, ezáltal lehetőség nyílik arra is, hogy több DPMI szolgáltatást használó program egymástól függetlenül, egyidőben fusson, továbbá az LDT a védelem szempontjából is nagyon fontos.

A lokális deszkriptorokat a 0. funkcióval lehet lefoglalni, és ha már nincs rá szükség, az 1. funkcióval szabadítható fel. Egyszerre akár több deszkriptor is lefoglalható, ez esetben a 3. funkcióra is szükség van azért, hogy pontosan megkaphassuk az egyes deszkriptorokat.

Lokális deszkriptorok lefoglalása		
Be	AX	0
	CX	a lefoglalni kívánt deszkriptorok száma
Ki	AX	a lefoglalt szelektor

Lokális deszkriptor felszabadítása		
Be	AX	1
	BX	a felszabadítani kívánt szelektor
Ki	–	

Több lefoglalt deszkriptor esetén a növelés lekérdezése		
Be	AX	3
Ki	AX	szám, amivel megkapjuk a következő szelektort

A deszkriptor lefoglalása után szinte bármilyen műveletet végrehajthatunk rajta: beállíthatjuk a báziscímét, a határát, és az elérési jogát. A báziscímet is lekérdezhetjük egy funkcióval, a többi értéket pedig a LAR és az LSL utasítással tudhatjuk meg.

Deszkriptor báziscímének lekérdezése		
Be	AX	6
	BX	szelektor
Ki	CX:DX	32 bites báziscím

Deszkriptor báziscímének beállítása		
	AX	7
Be	BX	szelektor
	CX:DX	32 bites báziscím
Ki	–	

Deszkriptor határának beállítása		
	AX	8
Be	BX	szelektor
	CX:DX	32 bites határ
Ki	–	

Deszkriptor elérési jogainak beállítása		
	AX	9
Be	BX	szelektor
	CX	elérési jogok
Ki	–	

A fenti néhány műveletet elvégezhetjük akár egy lépésben is, vagyis beállíthatjuk, illetve lekérdezhetjük egy deszkriptornak a báziscímét, határát, és jellemzőit egyetlen funkció hívásával.

Teljes deszkriptor lekérdezése		
	AX	11
Be	BX	szelektor
	ES:EDI	8 byte-os adatterület a deszkriptor számára
Ki	–	

Teljes deszkriptor beállítása		
	AX	12
Be	BX	szelektor
	ES:EDI	deszkriptort tartalmazó 8 byte-os adatterület címe
Ki	–	

A lokális deszkriptorokkal egyéb műveleteket is elvégezhetünk. Egy már meglévő lokális deszkriptort megduplázhatunk a 10. funkcióval, vagyis létrehozhatunk egy olyan adatszegmenset, amelynek a báziscíme és a határa megegyezik az eredeti deszkriptoréval. Ez elsősorban a kódszegmensek elérését könnyíti meg, mivel az új adatszegmens deszkriptor segítségével a kódszegmenseket is módosíthatjuk.

Ha egy program számára mindenképpen egy adott lokális deszkriptor szükséges (pl. a 0Fh szelektor), akkor a 13. funkcióval megkísérelhető a lefoglalás.

Deszkriptor másolat készítése		
Be:	AX	10
	BX	szelektor
Ki:	AX	új szelektor

Egy bizonyos deszkriptor lefoglalása		
Be	AX	13
	BX	kívánt szelektor
Ki	–	

A védett módú programokban gyakran szükség van az 1 MB alatti terület egy részének elérésére. Ehhez jelenleg számos funkciót kellene hívogatni, mire elérhetjük az adott szegmenset.

Ha például a képernyőre közvetlenül szeretnénk írni, le kell foglalni egy lokális deszkriptort (0. funkció), majd beállítani a 0b8000h báziscímet, 4000 byte határt, és a 92h elérési jogot (7–9. funkciók), vagy akár az egész deszkriptor is beállítható egyszerre a 12. funkcióval. Ennél sokkal egyszerűbb, ha a 2. DPMI funkcióval a képernyő valós módú címét (0b800h) alakítjuk át védett módú deszkriptorrá. Ezzel a funkcióval egy valós módnak megfelelő (64 KB méretű) szegmenset kapunk.

Valós módú szegmens deszkriptorrá alakítása		
Be	AX	2
	BX	valós módú szegmens
Ki	AX	lokális deszkriptor szelektora (64 KB határ)

A DPMI 1.0 kiszolgálóknál egyszerre akár több lokális deszkriptor is kezelhető a 11. és a 12. funkciókhoz hasonlóan. Ekkor az ES:EDI regiszterekben egy olyan terület címét kell átadni, amelynél minden deszkriptorhoz 10 byte tartozik. Az első 2 byte az adott deszkriptor szelektorát, a következő 8 byte pedig magát a deszkriptort tartalmazza. Ezzel például egy program kezdeti inicializálása egyetlen funkció hívásával is elvégezhető.

Több teljes deszkriptor lekérdezése		DPMI 1.0
	AX	14
Be	CX	szelektorok száma
	ES:EDI	speciális adatterület
Ki	–	

Több teljes deszkriptor beállítása		DPMI 1.0
	AX	15
Be	CX	szelektorok száma
	ES:EDI	speciális adatterület
Ki	–	

A következő program kiírja a képernyőre a parancssorban átadott paramétert, amely a PSP 81h címétől kezdve helyezkedik el, a paraméter hosszát pedig a 80h-n lévő byte jelzi. Mivel védett módba kapcsolás után az ES regiszter a PSP szelektorát tartalmazza, ezért ezt használjuk fel a paraméter megjelenítéséhez. A képernyőre íráshoz pedig a 2. funkcióval foglalunk le egy deszkriptort.

```
DPMI Segment Para Public Use16 'CODE'
    Assume cs:DPMI, ds:DPMI

SwitchPtr dd 0

Str_DPMI        db 'DPMI nem található.',10,13,'$'
Str_Switch      db 'Nem sikerült átkapcsolni védett '
                db 'módba.',10,13,'$'
Str_Param       db 'A paraméter:',0
Str_NoParam     db 'Nincs paraméter.',0


Start:
   push cs
   pop  ds

   mov  ax,3
   int  10h

   mov  ax,1687h              ; a DPMI host címének
   int  2fh                   ; lekérdezése
   or   ax,ax
   jnz  NoDPMI                ; nincs DPMI

   mov  word ptr SwitchPtr,di          ; ofszet cím
   mov  word ptr SwitchPtr+2,es        ; szegmens cím
```

```
    mov  ax,DPMI2               ; adatszegmens a host számára
    mov  es,ax
    xor  ax,ax                  ; 0. bit: 16 bites program
    call SwitchPtr              ; átkapcsolás védett módba
    jc   NoSwitch               ; nem sikerült...

ProtMode:

    push es                     ; a PSP szelektor az fs-be
    pop  fs

    mov  bx,0b800h              ; a videomemória szegmens
    mov  ax,2                   ; átalakítása deszkriptorra
    int  31h                    ; DPMI hívása
    jc   Exit                   ; ha hiba történt, kilépés

    mov  es,ax                  ; es:edi = képernyõ
    xor  edi,edi
    lea  esi,Str_Param          ; ds:esi=string
    mov  cl,fs:[80h]            ; cl=a paraméter hossza
    or   cl,cl                  ; van egyáltalán paraméter?
    jnz  L1                     ; nincs
    lea  esi,Str_NoParam

    mov  ah,7                   ; a szöveg színe
L1:
    lodsb
    stosw                       ; egy karakter kiírása
    or   al,al
    jne  L1                     ; folytatás 0-ig

    or   cl,cl
    jz   Exit                   ; ha nincs paraméter, kilépés

    mov  esi,81h                ; fs:esi=paraméter
L2:
    mov  al,fs:[esi]
    stosw                       ; egy karakter kiírása
    inc  esi                    ; következõ karakter
    dec  cl
    jnz  L2

Exit:
    mov  ax,4c00h               ; kilépés
    int  21h


NoDPMI:                         ; hibaüzenetek
    lea  dx,Str_DPMI
    jmp  Error
NoSwitch:
    lea  dx,Str_Switch
```

```
Error:
    mov   ah,9
    int   21h
    jmp   Exit


DPMI    Ends

DPMI2 Segment                   ; privát adatszegmens
     db 0ffffh dup (?)          ; a DPMI host számára
DPMI2 Ends

End Start
```

2.4. Memóriakezelő funkciók

A dinamikus memóriakezelés szinte minden nagyobb alkalmazás esetén szükséges. Természetesen a DPMI is tartalmaz olyan funkciókat, amelyek a kiterjesztett memória lefoglalására, felszabadítására, illetve átméretezésére szolgálnak a program futása közben. A szabad memória méretét az 500h funkcióval kérdezhetjük le. Az ES:EDI-ben egy 48 byte-os terület címét kell elhelyezni, és a DPMI hívása után a terület első duplaszava tartalmazza a szabad memóriát byte-okban.

Kiterjesztett memória foglalása		
Be	AX	501h
	BX:CX	a lefoglalni kívánt memória mérete byte-okban
Ki	BX:CX	a lefoglalt terület 32 bites lineáris címe
	SI:DI	azonosító

Kiterjesztett memória felszabadítása		
Be	AX	502h
	SI:DI	azonosító
Ki	–	

Lefoglalt EMB méretének megváltoztatása		
	AX	503h
Be	BX:CX	az új méret paragrafusokban
	SI:DI	a régi azonosító
Ki	BX:CX	a lefoglalt terület új lineáris címe
	SI:DI	az új azonosító

A memóriafoglaláskor csak egy lineáris címet kapunk, tehát ha használni szeretnénk ezt a területet, akkor előbb le kell foglalni egy lokális deszkriptort is, majd be kell állítani a báziscímet, a határt, és a deszkriptor típusát.

Lehetőség van az 1 MB alatti területek lefoglalására is, ez elsősorban a DOS eszközmeghajtók, és egyéb programokkal való kommunikációhoz szükséges, illetve ha a kívánt memória mérete olyan kicsi, hogy akár a 640 KB alatti területen is elfér.

DOS memória foglalása		
Be	AX	100h
	BX	a lefoglalni kívánt memória mérete paragrafusokban
	AX	a valós módú szegmenscím
Ki	BX	a legnagyobb lefoglalható méret
	DX	a lefoglalt memória szelektora

DOS memória felszabadítása		
Be	AX	101h
	DX	a felszabadítani kívánt szelektor
Ki	–	

Lefoglalt DOS memória méretének megváltoztatása		
	AX	102h
Be	BX	az új méret paragrafusokban
	DX	a lefoglalt memória szelektora
Ki	BX	ha nem sikerült, a legnagyobb lefoglalható méret

Mivel a DPMI kiszolgálóknál a lapozás engedélyezett, ezért a lineáris cím nem mindig egyezik meg a fizikai címmel. Az esetek többségében ez nem is okoz gondot, viszont előfordulhat, hogy szükség van egy bizonyos memóriaterület fizikai címének megfelelő lineáris címre. Ennek megvalósítására szolgál a 800h funkció.

Fizikai cím lineáris címmé alakítása		
	AX	800h
Be	BX:CX	fizikai cím
	SI:DI	memóriaterület mérete byte-okban
Ki	BX:CX	lineáris cím

Például az újabb videokártyák már támogatják a Linear Frame Buffert, amivel a teljes videomemória egy összefüggő területre osztható ki a kiterjesztett memóriában. Ugyanis a nagyobb felbontású grafikus módok számára már nem elegendő a 0A000h-nál lévő 64 KB-os (ill. 128 KB-os) terület. Ezt egy ideig a videomemória lapozásával oldották meg, ami rengeteg inkompatibilis videokártyához vezetett, mivel minden gyártó másképp valósította ezt meg.

A Linear Frame Buffer tehát szükségtelenné teszi ezt a lapozást, mivel az egész videomemória egyszerre elérhető. Ehhez viszont tudni kell az LFB fizikai címének megfelelő lineáris címet a DPMI alatt, erre szolgál a fenti funkció.

A videokártyán kívül persze több eszköz is van, amelyekhez DPMI alatt a fizikai-lineáris cím átalakítás szükséges.

2.5. Lapozással kapcsolatos funkciók

Azoknál a DPMI kiszolgálóknál, ahol a lapozás illetve a virtuális memória engedélyezve van, bármelyik pillanatban megtörténhet, hogy a memória egyes területei a fizikai memóriából lemezre kerülnek. Néhány program esetén szükség lehet arra, hogy bizonyos területek ne kerüljenek ki a fizikai memóriából, mivel az újabb betöltések miatti időveszteség problémákhoz vezethet.

Ennek az elllenkezője is előfordulhat, vagyis ha tudjuk, hogy egy adott területet (pl. egy nagyobb képet) hosszú ideig nem használunk, akkor jelezhetjük, hogy nincs rá szükségünk, és nyugodtan eltávolítható a memóriából. Az is lehet, hogy annyira felesleges az adott terület, hogy még lemezre sem érdemes írni, hanem egyszerűen felülírható. A következő néhány funkció ezeket valósítja meg.

Lapozás letiltása egy területen		
Be	AX	600h
	BX:CX	lineáris cím
	SI:DI	memóriaterület mérete byte-okban
Ki	–	

Lapozás engedélyezése újra egy területen		
Be	AX	601h
	BX:CX	lineáris cím
	SI:DI	memóriaterület mérete byte-okban
Ki	–	

Lapozás letiltása az 1 MB alatti területen		
Be	AX	602h
	BX:CX	lineáris cím
	SI:DI	memóriaterület mérete byte-okban
Ki	–	

Lapozás visszaállítása az 1 MB alatti területen		
Be	AX	603h
	BX:CX	lineáris cím
	SI:DI	memóriaterület mérete byte-okban
Ki	–	

Lapméret lekérdezése		
Be	AX	604h
Ki	BX:CX	egy lap mérete byte-okban

Ideiglenesen használatlan terület jelzése		
Be	AX	702h
	BX:CX	lineáris cím
	SI:DI	memóriaterület mérete byte-okban
Ki	–	

Felesleges terület jelzése		
Be	AX	703h
	BX:CX	lineáris cím
	SI:DI	memóriaterület mérete byte-okban
Ki	–	

2.6. A megszakításokkal kapcsolatos funkciók

Valós módban a megszakításvektorok módosítása elég egyszerű, csak a 0000:0000-n lévő vektorokat kell a megfelelő módon megváltoztatni, vagy a DOS funkciókat használni. Védett módban viszont az IDT akárhol lehet a memóriában, ráadásul nem is lenne biztonságos, ha bármelyik program kedve szerint átírhatná az IDT-t. A védett módú megszakítás-kezelők beállítása és lekérdezése DPMI alatt a következő két funkcióval végezhető el.

Megszakításvektor címének lekérdezése		
Be	AX	204h
	BL	a megszakítás sorszáma (0-255)
Ki	CX:EDX	a megszakítás-kezelő címe

Megszakításvektor címének beállítása		
	AX	205h
Be	BL	a megszakítás sorszáma
	CX:EDX	a megszakítás-kezelő címe
Ki	–	

Szükség lehet arra is, hogy a valós módú megszakítások címét kérdezzük le vagy módosítsuk. Ha DPMI alatt valós (ill. virtuális) módban vagyunk, akkor a 0000:0000 terület átírása, és a DOS 25h/35h funkció továbbra is használható. Védett módban ez a következő két funkcióval végezhető el az előzőekhez hasonlóan, de valós módú szegmens-, és ofszetcímekkel.

Valós módú megszakításvektor lekérdezése		
Be	AX	200h
	BL	a megszakítás sorszáma
Ki	CX:DX	valós módú megszakítás-kezelő címe (szegmens:ofszet)

Valós módú megszakításvektor beállítása		
Be	AX	201h
	BL	a megszakítás sorszáma
	CX:DX	a megszakítás-kezelő címe
Ki	–	

Bizonyos programok esetén fontos, hogy értesüljenek arról, ha valamilyen kivétel keletkezett, és azt saját maguk kezeljék le. Erre szolgál a 203h funkció,

amivel a védett módban bekövetkezett kivétel kezelője módosítható. Kivétel esetén persze nem közvetlenül az átirányított címre kerül a vezérlés, hanem előbb a DPMI kiszolgálóhoz, és majd ő hívja meg a mi kezelőnket.

Kivételkezelő címének lekérdezése		
Be	AX	202h
	BL	a kivétel sorszáma (0-31)
Ki	CX:EDX	a kivételkezelő címe

Kivételkezelő címének beállítása		
Be	AX	203h
	BL	a kivétel sorszáma
	CX:EDX	a kivételkezelő címe
Ki	–	

A DPMI 1.0 alatt lehetőség van a védett és a valós módú kivételkezelők lekérdezésére (210h és 211h funkciókkal) és beállítására is (212h illetve 213h).

A megszakításokkal kapcsolatban fontos kérdés az IRQ-k engedélyezése és letiltása is. Természetesen az nem engedhető meg, hogy egy DPMI alatt futó program teljesen letiltsa a megszakításokat, hisz akkor pre-emptiv multitaszking esetén a többi taszk nem kapna vezérlést, és a hardver eszközök sem generálhatnának megszakítást, stb.

Ezért az egyes taszkok csak lokálisan tilthatják le a megszakításokat, vagyis ezalatt csak számukra nem generálódnak IRQ-k, a többi taszkot ez nem érinti. A megszakításokat 3 DPMI funkcióval lehet irányítani, de használható továbbra is a CLI és az STI utasítások, amelyeket majd az operációs rendszer észrevétlenül emulál, és a taszkok számára lokálisan letiltja, illetve engedélyezi a megszakításokat.

Megszakítások letiltása		
Be	AX	900h
Ki	AL	korábbi állapot (0 = le volt tiltva, 1 = engedélyezve volt)

Megszakítások engedélyezése		
Be	AX	901h
Ki	AL	korábbi állapot (0 = le volt tiltva, 1 = engedélyezve volt)

Megszakítások állapotának lekérdezése		
Be	AX	902h
Ki	AL	állapot (0 = letiltva, 1 = engedélyezve)

2.7. Valós és védett mód közötti váltások

Gyakran szükség lehet arra, hogy védett módból meghívjunk egy valós módú megszakítást, például a BIOS-t, vagy az egér drivert, esetleg a TCP/IP packet drivert, stb. Ezt DPMI alatt a 300h funkcióval érhetjük el, meghatározva, hogy melyik regiszterben milyen érték legyen a megszakítás hívásakor. Szükség esetén megadható, hogy a DPMI kiszolgáló a védett módú veremből a paramétereket átmásolja a valós módú verembe. A megszakítás befejezése után megkapjuk a regiszterek új tartalmát is.

Valós módú megszakítás hívása		
	AX	300h
Be	BL	a megszakítás sorszáma (0–255)
	CX	a valós módú verembe átmásolandó szavak száma
	ES:EDI	valós módú regiszterértékek címe a memóriában
Ki	ES:EDI	az új regiszterértékek címe

A valós módú megszakításnak átadott értékeket nem a valódi regiszterekben kell elhelyezni, hanem egy 50 byte-os memóriaterületen, amelynek a címét az ES:EDI tartalmazza. A megszakítás végén ugyanezen a területen vannak a visszaadott regiszterek értékei. Az egyes regiszterek elhelyezkedése az alábbi ábrán látható.

Valós módú regiszterek másolata				
0h	EDI		ESI	
8h	EBP		0	
10h	EBX		EDX	
18h	ECX		EAX	
20h	Flags	ES	DS	FS
28h	GS	IP	CS	SP
30h	SS			

Mivel a CS és az IP értékét úgyis az adott megszakításhoz tartozó vektor határozza meg, ezért ezek a mezők figyelmen kívül hagyhatóak. A valós módú megszakítások általában csak 16 bites regisztereket használnak, ezért elegendő a 32 bites regiszterek alsó 16 bitjét beállítani. Az SS:SP regisztereket sem szükséges beállítani, ha ezek értéke 0, akkor a DPMI kiszolgáló automatikusan biztosít vermet a valós módú megszakítás számára.

Lehetőség van valós módú távoli eljárás hívására is védett módból. A 301h funkció egy FAR CALL-t hajt végre (visszatérés a RETF utasítással), a 302 funkció pedig a FAR CALL előtt a PUSHF-al a flag-eket is elmenti (visszatérés az IRET-el.)

A veremben átadott paraméternek itt van igazán jelentősége, hisz az eljárásoknál sokkal gyakoribb a vermen keresztüli paraméterátadás, mint a megszakításoknál. A valós módú regiszterek értékét az előzővel megegyező módon a memóriába másolva adhatjuk át, és ugyanitt kaphatjuk meg az eljárás hívása utáni értéket. Itt már a CS és az IP tartalmát is helyesen be kell állítani az eljárás valós módú címére.

Valós módú távoli eljárás hívása		
	AX	301h
Be	CX	a valós módú verembe átmásolandó szavak száma
	ES:EDI	valós módú regiszterértékek címe a memóriában
Ki	ES:EDI	az új regiszterértékek címe

Valós módú távoli eljárás hívása PUSHF után		
	AX	302h
Be	CX	a valós módú verembe átmásolandó szavak száma
	ES:EDI	valós módú regiszterértékek címe a memóriában
Ki	ES:EDI	az új regiszterértékek címe

Ekkor van igazán jelentősége a valós módú megszakításvektorok és kivételkezelők beállításának (201h és 203h funkciók), mert amikor egy valós módú eljárást, vagy egy megszakítást hívunk meg, akkor könnyen előfordulhat, hogy például egy IRQ, vagy kivétel keletkezik, és ekkor szükség lehet a megszakítások megfelelő kezelésére.

Nemcsak védett módból lehet valós módú eljárást meghívni, hanem fordítva is, valós módból is meghívható védett módú eljárás. Ehhez védett módban le kell foglalni egy ún. callback címet (CX:DX-ben), amit átadhatunk a valós módú programnak, amely majd „visszahívja" az eljárásunkat (FAR CALL).

Védett módú távoli eljárás hívása		
	AX	303h
Be	DS:ESI	védett módú eljárás címe
	ES:EDI	a valós módú regiszterek képe a memóriában
Ki	CX:DX	a valós módú cím

Védett módú távoli eljárás címének felszabadítása		
	AX	304h
Be	CX:DX	a lefoglalt valós módú cím
Ki	–	

Arra valószínűleg nem gondoltak, hogy az adott eljárás lehet egy csak olvasható kódszegmensben is, amit a DS-be nem lehet betölteni, mert általános védelmi hiba keletkezik.

A megszakítások és eljárások hívásán kívül lehetőség van arra is, hogy közvetlenül átkapcsoljunk a másik üzemmódba.

Közvetlen módváltások címeinek lekérdezése		
Be	AX	306h
Ki	BX:CX	valós módból védett módba kapcsolás címe
	SI:EDI	védett módból valós módba kapcsolás címe

A visszaadott címeket egy FAR JMP-vel kell meghívni, a következő értékekkel:

AX	DS új értéke
CX	ES új értéke
DX	SS új értéke
EBX	ESP új értéke
SI	CS új értéke
EDI	EIP új értéke

Valós módba kapcsoláskor értelemszerűen a 32 bites regiszterek helyett a 16 biteseket kell használni.

A módváltás meghívása előtt célszerű elmenteni az aktuális állapotot is (regiszterek, stb), majd később visszaállítani. Ez megoldható a 305h funkció segítségével, amely egy valós és egy védett módú távoli címet ad vissza.

Az állapotmentés eljárás címének lekérdezése		
Be	AX	305h
	AX	az elmentéshez szükséges memória mérete
Ki	BX:CX	valós módú elmentés címe
	SI:EDI	védett módú elmentés címe

Ezeket a címeket kell egy FAR CALL-al meghívni, a következő értékekkel:

AL	0 = állapot mentés, 1 = állapot visszaállítás
ES:EDI	memóriaterület a mentés ill. visszaállítás számára

Az ES:EDI-ben lévő memóriaterület méretét a 305h funkcióval kapott AX értéke határozza meg. Ha ez nulla, akkor az adott DPMI kiszolgáló esetén nincs szükség az állapot elmentésére és visszaállítására a közvetlen módváltás előtt és után, és ekkor a távoli eljárások hívása is hatástalan.

Védett módban gyakran szükség lehet a valós módú megszakítások hívására, főleg az olyan funkciók elvégzéséhez, amelyeket védett módban nem, vagy csak nehezen lehetne megvalósítani. A következő példában 3 valós megszakítást hívunk meg: először töröljük a képernyőt a BIOS segítségével (INT 10h), majd kiírunk egy szöveget a 9-es DOS funkcióval (INT 21h), végül egy billentyű lenyomására várunk (INT 16h).

A 300h funkcióhoz szükséges regisztereket az IntRegs területén helyezzük el, és ennek címét adjuk meg az ES:DI-ben az INT 31h hívása előtt. A valós módú megszakítások hívása tehát nagyon egyszerűen elvégezhető a DPMI funkciók használatával, és hasonló módon akár valós módú eljárások is meghívhatóak. Ezáltal lehetőség van az összes fontos művelet (pl. filekezelés) elvégzésére is védett módban.

```
.386
.model small
.stack

DPMI Segment Para Public Use16 'CODE'
    Assume cs:DPMI, ds:DPMI

SwitchPtr  dd 0
Str_DPMI   db 'DPMI nem található.',10,13,'$'
Str_Switch db 'Nem sikerült átkapcsolni védett '
           db 'módba.',10,13,'$'

Str_Ex3    db '3. DPMI példa: Valós módú '
           db 'megszakítások hívása',10,13,10,13
           db 'Nyomj meg egy billentyût!',10,13,7,'$'

; A regiszterek képe a 300h DPMI funkcióhoz

IntRegs Label   Dword
_edi      dd        0          ; 0
_esi      dd        0          ; 4
_ebp      dd        0          ; 8
_res      dd        0          ; 0ch
_ebx      dd        0          ; 10h
_edx      dd        0          ; 14h
_ecx      dd        0          ; 18h
_eax      dd        0          ; 1ch
_flags    dw        0          ; 20h
_es       dw        0          ; 22h
_ds       dw        0          ; 24h
_fs       dw        0          ; 26h
_gs       dw        0          ; 28h
_ip       dw        0          ; 2ah
_cs       dw        0          ; 2ch
_sp       dw        0          ; 2eh
_ss       dw        0          ; 30h

Start:
   push cs
   pop  ds
   cld

   mov  ax,1687h               ; a DPMI host címének
   int  2fh                    ; lekérdezése
   or   ax,ax
   jnz  NoDPMI                 ; nincs DPMI

   mov  word ptr SwitchPtr,di          ; ofszet cím
   mov  word ptr SwitchPtr+2,es        ; szegmens cím
```

```
    mov  ax,DPMI2               ; ES=adatszegmens a host
    mov  es,ax                  ; számára
    xor  ax,ax                  ; 0. bit: 16 bites program
    call SwitchPtr              ; átkapcsolás védett módba
    jc   NoSwitch               ; nem sikerült...

ProtMode:

    mov  ax,ds
    mov  es,ax                  ; ES = adatszegmens

    mov  _ds,DPMI               ; adatszegmens beállítása
    mov  _es,DPMI
    mov  _ss,DPMI_S             ; a verem beállítása
    mov  _sp,4096

; INT 10h, AX=3 hívása (képernyõtörlés)

    mov  _eax,3

    mov  ax,300h                ; 300h funkció
    mov  bx,10h                 ; 10h megszakítás hívása
    lea, di,IntRegs             ; ES:DI = a regiszterek képe
    mov  cx,0                   ; nincs paraméter
    int  31h                    ; DPMI hívása

; INT 21h, AX=9 hívása (string kiírása)

    mov  _eax,900h              ; ah = 9
    mov  _edx,offset Str_Ex3    ; a string címe
    mov  ax,300h                ; 300h funkció
    mov  bx,21h                 ; 21h megszakítás hívása
    lea  di,IntRegs             ; ES:DI = a regiszterek képe
    mov  cx,0                   ; nincs paraméter
    int  31h                    ; DPMI hívása

; INT 16h, AX=0 hívása (várakozás egy billentyûre)

    mov  _eax,0                 ; 0. funkció

    mov  ax,300h                ; 300h funkció
    mov  bx,16h                 ; 16h megszakítás hívása
    lea  di,IntRegs             ; ES:DI = a regiszterek képe
    mov  cx,0                   ; nincs paraméter
    int  31h                    ; DPMI hívása

Exit:
    mov  ax,4c00h               ; kilépés
    int  21h
```

```
NoDPMI:                                 ; hibaüzenetek
    lea   dx,Str_DPMI
    jmp   Error
NoSwitch:
    lea   dx,Str_Switch
Error:
    mov   ah,9
    int   21h
    jmp   Exit


DPMI    Ends

DPMI2 Segment
db    0ffffh dup (?)
DPMI2 Ends

DPMI_S Segment
dd    1024 dup (?)                      ; 4K verem (nem szükséges)
DPMI_S Ends

End Start
```

2.8. DPMI hibakódok

Ha a DPMI funkciók hívásakor bármilyen hiba történik, akkor a Carry flag mindig 1, egyébként 0. Az 1.0-s DPMI kiszolgálók hiba esetén mindig visszadnak az AX-ben egy hibakódot is, a 0.9-es verziójú kiszolgálók pedig általában nem.

Az eddigiek során az egyes funkcióknál esetlegesen előforduló hibakódok nem szerepeltek külön, hanem inkább egy helyen, az alábbi összefoglaló táblázatban találhatóak meg.

Hibakód	Megnevezés
0007H	Valós módú memóriaterület foglaltsága megsérült
0008H	Nincs elég valós módú memória
0009H	Érvénytelen memóriaszegmens paraméter
8001H	Érvénytelen funkció, a kiszolgáló által nem támogatott, vagy ismeretlen funkció hívása
8002H	Érvénytelen állapot, a funkcióhoz szükséges valamely elem nem megfelelő állapotban van
8003H	A kívánt funkció veszélyeztetné a rendszer épségét
8004H	Holtpont kialakulásának veszélye
8005H	Kérés megtagadva
8010H	Erőforrás nem elérhető, a DPMI kiszolgáló nem tudja lefoglalni a funkcióhoz szükséges erőforrást
8011H	Deszkriptor nem elérhető, nem lehet lefoglalni
8012H	Lineáris memória nem elérhető
8013H	Fizikai memória nem elérhető
8014H	Háttértároló nem elérhető
8015H	Callback nem elérhető
8016H	Azonosító nem elérhető
8017H	Túl sok memória zárolás
8018H	Az erőforrás már kizárólagosan le van foglalva
8019H	Az erőforrás már megosztottan le van foglalva
8021H	Érvénytelen szám
8022H	Érvénytelen szelektor, a paraméter nem érvényes deszkriptorra hivatkozik.
8023H	Érvénytelen azonosító
8024H	Érvénytelen callback
8025H	Érvénytelen lineáris cím
8026H	Érvénytelen művelet

2.1. ábra A DPMI hibakódok

ASSEMBLY
VÉDETT MÓDÚ
PROGRAMOZÁS
LEMEZMELLÉKLET

1600.

ISBN 963-577-199-1

PRESENT®
COMPUTER Számítástechnikai
Kereskedés és Szoftverház
1065 Budapest, Nagymező u. 64. Telefon: 132-77-51

PC
PINCE